NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ADVANCED NAVAL SURFACE FIRE SUPPORT WEAPON EMPLOYMENT AGAINST MOBILE TARGETS

by

Hung B. Le

December 1999

Thesis Advisor: Arnold H. Buss
Second Reader: Douglas J. MacKinnon

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1999	Master's Thesis

4. TITLE AND SUBTITLE
ADVANCED NAVAL SURFACE FIRE SUPPORT WEAPON EMPLOYMENT AGAINST MOBILE TARGETS

5. FUNDING NUMBERS

6. AUTHOR(S)
Le, Hung B.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
The Johns Hopkins University Applied Physics Laboratory
11100 Johns Hopkins Road
Laurel, Maryland 20723-6099

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Key threat trends have identified shortfalls in Naval Surface Fire Support (NSFS), a mission area that is undergoing rapid evolution. The Navy's ability to effectively provide sea-based fire support to ground forces is profoundly challenged by mobile and reduced dwell time targets. Furthermore, longer range enemy weapon systems, which must be destroyed at greater ranges prior to their engagement of friendly forces, will make NSFS timeliness a difficult proposition. To overcome these threat trends, the United States is developing sophisticated weapons that promise increased lethality, greater ranges and improved responsiveness. However, the development of robust firing policies to ensure effective weapon utilization has lagged behind the hardware. Existing computer models and simulations have not addressed the question of NSFS gun/missile firing policy. This thesis develops the Naval Surface Fire Support Simulation (NSFSSim) model, a discrete-event simulation that serves as an analysis tool to determine favorable firing policies for future NSFS gun and missile systems in support of determining the appropriate NSFS weapons mix. NSFSSim models ships and their associated NSFS weapons in counterbattery and call fire missions against mobile, reduced dwell time targets. Exploratory analysis using NSFSSim yields useful insights, and the component-based architecture underlying the model provides significant flexibility for further analysis.

14. SUBJECT TERMS
Discrete-Event Simulation, Firing Policy, Java, Modeling and Simulation, Naval Surface Fire Support

15. NUMBER OF PAGES
103

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239.18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

ADVANCED NAVAL SURFACE FIRE SUPPORT WEAPON EMPLOYMENT AGAINST MOBILE TARGETS

Hung B. Le
Lieutenant, United States Navy
B.S., United States Naval Academy, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
December 1999

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Key threat trends have identified shortfalls in Naval Surface Fire Support (NSFS), a mission area that is undergoing rapid evolution. The Navy's ability to effectively provide sea-based fire support to ground forces is profoundly challenged by mobile and reduced dwell time targets. Furthermore, longer range enemy weapon systems, which must be destroyed at greater ranges prior to their engagement of friendly forces, will make NSFS timeliness a difficult proposition. To overcome these threat trends, the United States is developing sophisticated weapons that promise increased lethality, greater ranges and improved responsiveness. However, the development of robust firing policies to ensure effective weapon utilization has lagged behind the hardware. Existing computer models and simulations have not addressed the question of NSFS gun/missile firing policy. This thesis develops the Naval Surface Fire Support Simulation (NSFSSim) model, a discrete-event simulation that serves as an analysis tool to determine favorable firing policies for future NSFS gun and missile systems in support of determining the appropriate NSFS weapons mix. NSFSSim models ships and their associated NSFS weapons in counterbattery and call fire missions against mobile, reduced dwell time targets. Exploratory analysis using NSFSSim yields useful insights, and the component-based architecture underlying the model provides significant flexibility for further analysis.

DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF SYMBOLS, ACRONYMS AND/OR ABBREVIATIONS

AOA	Analysis of Alternatives
ASM	Anti-Ship Missile
ATACMS	Army Tactical Missile System
C^2	Command and Control
C^4I	Command, Control, Communications, Computers, and Intelligence
CEP	Circular Error Probable
ELAN	Enhanced Lanchester
ERGM	Extended Range Guided Munition
FFTS	Forward...From The Sea
FO	Forward Observer
GPS	Global Positioning System
GUI	Graphical User Interface
IOC	Initial Operational Capability
ITEM	Integrated Theater Engagement Model
JHU/APL	Johns Hopkins University, Applied Physics Laboratory
LASM	Land Attack Standard Missile
MEF	Marine Expeditionary Force
MK	Mark
MOE	Measure of Effectiveness
MOP	Measure of Performance
MRSI	Multiple Rounds Simultaneous Impact
MVC	Model-View-Controller
NGFS	Naval Gun Fire Support
NPS	Naval Postgraduate School
NSFS	Naval Surface Fire Support
NSFSSim	Naval Surface Fire Support Simulation
NTACMS	Navy Tactical Missile System
OMFTS	Operational Maneuver From The Sea
OOP	Object-Oriented Programming
OPNAV	Office of the Chief of Naval Operations
PK	Probability of Kill
RSTA	Reconnaissance/Surveillance/Target Acquisition
SACC	Supporting Arms Coordination Center
SAM	Surface to Air Missile
SCLAWS	Surface Combatant Land Attack Weapons Study
SM	Standard Missile
SPA	Self-Propelled Artillery
TACAIR	Tactical Aircraft
TAFSM	Target Acquisition Fire Support Model
TBM	Theater Ballistic Missile
TLAM	Tomahawk Land Attack Missile
TLE	Target Location Error
TOF	Time of Flight

TPM	Technical Performance Measure
TTP	Tactics, Techniques, and Procedures
TTWS	Tactical Tomahawk Weapon System
VLS	Vertical Launching System

EXECUTIVE SUMMARY

Key threat trends have identified shortfalls in Naval Surface Fire Support (NSFS), a mission area that is undergoing rapid evolution. The Navy's ability to effectively provide sea-based fire support to ground forces is profoundly challenged by mobile and short dwell time targets. Furthermore, longer range enemy weapon systems, which must be destroyed at greater ranges prior to their engagement of friendly forces, will make NSFS timeliness a difficult proposition. To overcome these threat trends, the United States is developing sophisticated weapons that promise increased lethality, greater ranges and improved responsiveness. However, the development of robust firing policies to ensure effective weapon utilization has lagged behind the hardware. The fiscal reality of budgetary constraints and the challenges posed by ever-increasingly capable and mobile enemy weapon systems highlight the need for sound analysis in the area of tactical employment of precision weapons.

Existing computer models and simulations have not addressed the question of NSFS gun/missile firing policies. Some studies conducted to address other NSFS issues have successfully used a consortium-of-models approach. However, due to the rigid design of these simulation models, major modification to existing code is required to enable the models to work together. To overcome these difficulties, this thesis developed the Naval Surface Fire Support Simulation (NSFSSim) model, a component-based, discrete-event simulation that serves as an analysis tool to determine favorable firing policies for future NSFS gun and missile systems. While no single model can properly analyze all aspects of the complex problem of sea-based fire support, it can yield useful insights to a small portion of the larger problem.

NSFSSim runs on any hardware platform and can be easily modified to support additional features and greater resolution. This simulation model combines newly developed components with a few previously developed components. A graphical user interface was built to enable rapid modification of input data, execution of simulation runs with different views, and the immediate display of output that lends itself to analysis using operations research methods. Together, these components provide a useful analysis tool that is dynamic, flexible, and component based. The notional scenario presented in this thesis is designed to demonstrate the type of analysis that can be conducted using NSFSSim.

NSFSSim was created as a first step toward the goal of providing military planners and analysts with a component-based simulation tool that can aid in the formulation of integrated NSFS gun and missile firing policies against mobile/relocatable targets. Its uses extend beyond the analysis of firing sequences and dispense diameters undertaken thus far. The model can be used to investigate optimal artillery battery tactics against advanced NSFS weapons as well as the impact of response times, target location errors, and weapon precision limits on the success of NSFS missions. Component modifications and additions can be made easily to create future versions of NSFSSim that are more complex and robust.

ACKNOWLEDGMENT

I would like to thank Professor Arnold H. Buss for his outstanding guidance, assistance, and support throughout the thesis process. I also wish to thank Jack Keane of the Joint Warfare Analysis Department (JWAD) at JHU/APL for his dedicated efforts as thesis tour liaison at APL. I would also like to express my appreciation for the encouragement and insight given by the analysts of JWAD's Joint Theater Analysis Group—in particular, Ted Smyth, Richard Miller, and Alan Zimm. Furthermore, I want to thank Russ Gingras and Steve Biemer who provided for and sponsored me on my experience tour.

Last, but never least, I want to thank my wife Lyn, whose patience and understanding helped me through two demanding years at NPS.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The end of the Cold War has redefined the environment in which the Navy must operate. Amidst the challenges presented by increasingly scarce resources, the Navy has undergone a gradual metamorphosis from a "blue water" force developed for open-ocean engagements against the former Soviet Union to a littoral force that faces many potential adversaries. Today's Navy primarily projects power from the sea as an integrated part of Joint strike operations and in support of the Joint land battle. The experiences of Desert Shield/Desert Storm highlight the emerging prominence of naval support of ground forces.

The Navy's *Forward...From The Sea* (FFTS) and the Marine Corps' *Operational Maneuver From The Sea* (OMFTS), the Services' authoritative statements on warfighting, envision an expanded role for naval fire support in future operations. Similarly, *Joint Vision 2010* provides an operational template for future Joint warfighting that focuses on leveraging technology to achieve such concepts as precision engagement and dominant maneuver. Evolving warfighting concepts as well as advancements in weapons technologies have altered perceptions about and broadened the potential requirements for sea-based fire support. OMFTS, in particular, proposes dynamic strategies and tactics aimed at decisive action, mobility, surprise, and fires to enable maneuvers that exploit enemy weaknesses. Effective naval fire support is paramount if OMFTS is to be realized.

Historically, naval firepower from surface combatants has contributed to the success of nearly all military operations in or near the littorals. Traditional Naval Gun

Fire Support (NGFS) has encompassed all naval guns from 3-inch to 16-inch to support amphibious operations. Today's modern warships have either one or two Mark (MK) 45 5-inch/54-caliber guns capable of firing ballistic rounds to a maximum range of approximately thirteen nautical miles (nm). When precision fires are required, however, the maximum effective range becomes greatly reduced. Moreover, fire support planning and a plotting team using voice-reporting procedures is still accomplishing coordination on the most modern cruisers and destroyers. Similarly, the Supporting Arms Coordination Center (SACC) on the newest amphibious assault ship still employs the manual practices and procedures reminiscent of World War II fire support planning. Clearly, current weapon ranges, organization, and planning and coordination procedures are inadequate to meet the requirements of 21st century warfighting concepts.

The precepts of attrition warfare are being replaced by the Marine Corps' concept of maneuver warfare, a paradigm which "envisions a faster-paced, longer-range insertion of troops with greater reliance on naval fire support and logistics." (Allen, 1996) No longer viewed as a gun preparing a hostile beachhead for amphibious operations, offshore fire support in the near future will be provided by precision-guided munitions and tactical land attack missiles. These advanced weapons will be capable of destroying targets at ranges in excess of 100 nm.

In recognition of these changes and the expanded role of surface combatants in support of the Joint land battle, the Navy has updated its terminology, replacing NGFS with Naval Surface Fire Support (NSFS). Joint Pub 1-02, *Department of Defense Dictionary of Military and Associated Terms*, defines NSFS as "fires provided by Navy

surface gun, missile, and electronic warfare systems in support of a unit or units tasked with achieving the Joint commander's objective." (OCJCS, 1994)

Today NSFS is still limited in duration and is used primarily to provide short-range fire support until organic artillery assets are established ashore. Due to weapons limitations, fires are directed mainly at fixed defenses. In the near future, however, NSFS will be provided at greater ranges and for extended durations. In the early stages of the battle, sea-based fire support will serve as a surrogate for organic artillery, thereby enabling ground forces to maneuver against the enemy. Later in the battle, NSFS will complement tactical aircraft (TACAIR) and organic artillery ashore. Currently, logistics support and command and control (C^2) functions shift from sea to shore following the post-assault phase of an amphibious operation. In the future it is likely that these functions will remain offshore for as long as the situation permits. Advanced capability NSFS weapons are one of the primary enabling factors of this new operational concept.

These new weapons will include the Extended Range Guided Munition (ERGM) that will be fired from an improved 5-inch gun, a tactically employed Tomahawk missile, and a responsive land attack missile that uses an existing missile airframe. Each missile will compete for space inside shipboard MK 41 Vertical Launching Systems (VLS). All of these advanced weapons will utilize Global Positioning System (GPS) satellites for guidance to their respective aim points and promise greater lethality, range, and improved responsiveness.

Several key threat trends have generated the need for such sophisticated weapons. Chief among these trends are the improved mobility of artillery, theater ballistic missiles (TBMs), and surface to air missiles (SAMs) and the use of shorter dwell times. Improved

mobility and shorter dwell times equate to a reduced window of opportunity for fire support weapons to detect, acquire, and effectively engage enemy targets. The prospect of destroying such targets becomes especially remote because weapon times of flight (TOF) increase as a result of extended ranges. However, longer-range enemy weapon systems induce these extended ranges because the weapon systems must be destroyed prior to their engagement of friendly forces. Additionally, improved enemy deception capabilities will adversely affect friendly reconnaissance/surveillance/target acquisition (RSTA) sensor performance.

Naval surface-launched weapon systems are being developed to provide Aegis cruisers and destroyers the expanded capability of rapidly and precisely placing ordnance on target in support of the Joint land battle as well as expeditionary operations in the littorals. While weapons development has proceeded with the momentum of adequate funding, weapons systems integration and tactical considerations remain at the conceptual stages.

A. NSFS WEAPONS

The NSFS Program Office (PMS-429) of the Naval Sea Systems Command is developing the Ex-171 ERGM that will be fired from a modified 5-inch/62-caliber gun. The ERGM, which advertises a maximum range of 63 nm, is scheduled to be deployed on DDG 81 and later *Arleigh Burke* class destroyers in 2002. Subsequently, this gun system and the capability to fire ERGM will be backfitted on VLS-capable *Ticonderoga* class cruisers, specifically CG 52 and later ships. This enhanced munition will dispense bomblets using a variable dispense diameter feature. With this most important capability,

bomblet patterns can be concentrated to maximize lethality against a single target or broadened to allow the possibility of multiple mission kills against dispersed targets.

Tomahawk Land Attack Missiles (TLAMs) have already proved their effectiveness in strike missions against fixed defenses. NSFS integration of this potent weapon system involves the development of tactically tasked Tomahawk variants that are capable of in-flight retargeting in response to fire mission adjustments. The Baseline IV Tactical Tomahawk Weapon System (TTWS) will enable sea-based land attack ranges of 200 to 1,600 nm. The major operational requirements of TTWS are the following (JCM-2237, 1998):

- Increase system flexibility to support receipt of missile/mission communications and enroute retargeting of the missile to alternate preplanned outcome or emergent target
- Reduce system response time to allow engagement of emergent and relocatable targets
- Improve lethality against a wider target set
- Retain all Baseline III system capabilities (unless specifically exempted)

Required, but still unfunded, is a more responsive land attack missile adapted from an existing missile airframe. Two major candidate airframes exist. The first is the Standard Missile (SM-2), a capable but aging air defense missile employed on many surface combatants. In its modified NSFS role, the Land Attack Standard Missile (LASM) would carry a 120-pound improved unitary warhead and possess a maximum

range of 120 nm. The second candidate missile is the Army Tactical Missile System (ATACMS). The Navy version of this missile system, Navy Tactical Missile System (NTACMS), would carry a larger warhead weighing 390 pounds and extend the maximum range to 150 nm.

In *A National Security Strategy for a New Century*, President William J. Clinton states that "the military challenges of the 21st century, coupled with the aging of key elements of the U.S. force structure, require a fundamental transformation of our forces." One example of this transformation is the development of DD 21, the 21st Century Land Attack Destroyer, which has an Initial Operational Capability (IOC) date of 2008. Designed to replace *Oliver Hazard Perry* FFG 7 class frigates and *Spruance* DD 963 class destroyers, DD 21 will be a multi-mission platform. Its most potent mission, however, will be land attack warfare. The twenty-three planned DD 21 class destroyers will possess either a trainable or vertical 155-millimeter (mm) gun capable of firing 155-mm howitzers and larger versions of ERGM to ranges in excess of 100 nm. DD 21 will enjoy larger magazine capacities than today's Aegis cruisers and destroyers, making it even more formidable as an NSFS platform. It will also possess TTWS and a complementary land attack missile.

B. MOTIVATION

Advanced NSFS weapons will bring vast performance improvements over the current NSFS weapon, the MK 45 gun. Such technological sophistication comes with a heavy cost penalty, however. These weapons will be much more expensive than today's 5-inch ballistic ammunition. Cost concerns over ERGM have already surfaced. Recently

the Navy appointed an outside assessment team at the Massachusetts Institute of Technology's Lincoln Lab to examine the program. "Some Navy officials are concerned that the system's complexity may increase its cost and delay deployment, currently scheduled for 2002." (Holzer, 1999) Moreover, ship magazines will accommodate fewer of these larger munitions. Larger costs per weapon, fewer weapons per surface combatant, and the desire for efficiency motivate an investigation into optimality considerations for these advanced weapons.

The expectations for NSFS are at an all-time high. Sound qualitative and quantitative analyses must be conducted to support efficient acquisition decisions that meet emerging NSFS requirements. Similarly, analyses must be performed that investigate procedures and doctrine for the effective tactical employment of these advanced NSFS weapons. Existing computer models and simulations have not addressed the question of NSFS gun/missile firing policies. While no one-model approach can properly analyze all aspects of the complex problem of sea-based fire support, a single model alone can yield useful insights to a small portion of the larger problem.

The previous section suggested problems that mobile, short dwell time targets pose for NSFS weapons. To appreciate these problems, consider enemy weapon systems such as artillery guns and howitzers. Most modern self-propelled artillery (SPA) and towed artillery systems are capable of cross-country speeds of 40 or more kilometers per hour (km/hr). Recall that advanced NSFS weapons such as ERGM and LASM fly to an aim point believed to be the location of an enemy target. Because the aim point is determined prior to weapon launch and remains fixed, any movement by the target away from the aim point minimizes the likelihood of the weapon's impacting the target. A

Mach 2.0 LASM fired from a surface combatant stationed 25 nm from the enemy coast, against a moving artillery unit that is 25 nm inland, surely will miss. Traveling at speeds below Mach 1.0, an ERGM fired under the same conditions has no chance of success.

Realistically, an NSFS weapon can achieve a mission kill against a mobile target only during the target's dwell time, or the time that it remains stationary at a geographic location. The window of opportunity for achieving this mission kill likely is narrow for artillery systems. Conceivably, a SPA gun could take as little as 90 seconds to emplace or make preparations to fire its gun, could fire six rounds at the rate of six rounds per minute for a total of one minute, and take another 30 seconds to displace before moving to a new location. This tactic of firing rounds and then moving away from an aim point in avoidance of counterfire is commonly called "shoot and scoot." (Zimm, 1996) This particular artillery gun, then, would present a window of opportunity of three minutes for an incoming NSFS missile or munition that must travel upwards of 50 nm prior to impacting the aim point.

This thesis will develop the Naval Surface Fire Support Simulation (NSFSSim) model, a discrete-event simulation model that can provide useful insights into the problem of NSFS gun/missile firing policies against relocatable targets. The simulation model will be used to explore the following questions:

- In the tactical employment of ERGM and a land attack missile, what firing policies optimize mission effectiveness against mobile and short dwell time targets such as SPA and towed artillery batteries that utilize "shoot and scoot" tactics? Specifically, what gun/missile firing sequence(s) minimize the number of rounds fired by a given mix of artillery batteries?

- For a given mix of SPA and towed artillery batteries, is there an optimal ERGM dispense diameter (20 meters (m), 40 m, 60 m, 80 m, 100 m)?

NSFSSim is simple and does not profess to offer any definitive results. However, the model does provide some useful insights into the questions listed above. Combat is a complex and uncertain proposition. In this case, the uncertainty is compounded by the inclusion of future weapons systems, whose technical performance measures (TPM) are still evolving. While these unknown parameters introduce uncertainties in any model, they offer an open invitation for the application of simulation modeling. An analysis surrounding the questions posed in the previous paragraph is presented in Chapter III.

C. BACKGROUND

Studies have been performed to investigate the expanded role of surface combatants in support of land attack warfare. (Zimm, 1998) Analyses of alternatives have been conducted to evaluate the effectiveness of various land attack gun systems. (Zimm, 1999) Similarly, studies have been performed in efforts to decide which NSFS missiles should be installed on the Navy's newest surface combatants. (Schweizer, 1999) Spreadsheet optimization to determine optimal ship ordnance loadouts for NSFS missions has also been performed. (Chien, 1997) The impetus for these studies has been the evolving relationship of NSFS to the ground war as well as emerging weapons technologies. Prior to these analyses, the Office of the Chief of Naval Operations (OPNAV) Strike and Fire Support Branch of the Surface Warfare Division (N863F), along with the Amphibious Branch of the Expeditionary Warfare Division (N853), tasked

the Johns Hopkins University, Applied Physics Laboratory (JHU/APL) with developing a Road Map for NSFS. (Allen, 1996)

This Road Map was "defined as a time-phased summary of systems, concepts and issues critical to development of an acquisition plan" that extends through 2010 and beyond. (Allen, 1996) Phase 1 of the two-phase study provides a preliminary Road Map and was completed in 1996; Phase 2, which concentrates on the qualitative factors of NSFS and modeling NSFS' impact on the Joint land battle, is currently ongoing at JHU/APL.

The overall Road Map development in Phase 1 resulted in general observations, conclusions, and recommendations for the future of NSFS. Some of the observations on the current state of NSFS are:

- Perceptions have shifted from NGFS to NSFS.
- Warfighting concepts and scenarios are not yet mature.
- Joint command, control, communications, computers, and intelligence (C^4I) architectures are not keeping pace with weapons development.
- The organizational hierarchy established to manage NSFS architecture or "system-of-systems" is widely diffused.

Compounding these observations are key threat target trends that reveal shortfalls in NSFS and serve as drivers for future requirements. Among these trends are use of short dwell time and mobile targets and enemy employment of longer range weapon

systems. Chief among the conclusions and recommendations drawn from Phase 1 of the Road Map are (Allen, 1996):

- There exists a need for a new vision that captures the relationship between tactical and strategic fires and the key performance parameters of NSFS (range, lethality, and responsiveness).

- There is a need for quantitative and qualitative analyses to support sound Navy acquisition decisions.

In February 1998, JHU/APL released a report entitled *Land Attack Warfare Technical Studies* that addressed the above recommendations. The report documents the results of three studies conducted at JHU/APL. The first two investigations were performed under the umbrella of the Surface Combatant Land Attack Weapons Study (SCLAWS). The first was "a study which investigated the potential importance of Naval Surface Fire Support advanced gun weapon systems in the context of a Marine Expeditionary Force (MEF) level Joint-approved scenario." (Zimm, 1998) The second was a "study which investigated some of the issues surrounding optimizing the employment of low-Circular Error Probable (CEP) rounds." (Zimm, 1998) The third study investigated "the potential of using advanced Tactics, Techniques, and Procedures (TTP) in the employment of advanced NSFS weapons." (Zimm, 1998) All three investigations utilized a group of existing models, with and without major code modifications.

SCLAWS Part 1A concluded that ERGM is able to shape the battlefield prior to engagements through a superior combination of range, lethality, and responsiveness. In

addition, different munition types are necessary to effectively engage a diversity of targets. Target mobility and hardness issues were addressed. Lastly, the study concluded that surface combatants armed with anti-armor terminally homing rounds would benefit by preserving their ability to save ammunition for other targets. (Zimm, 1998)

Part 1B of SCLAWS was a weapons optimization analysis. Among the recommendations offered was the importance of target location error (TLE) reduction to improve fire support weapons effectiveness. (Zimm, 1998) Also recommended was the development of algorithms to determine optimal dispense diameters against different targets. The study determined that optimal dispense diameters vary for individual target types, but simulation runs with mixtures of different target types were not conducted.

The study also cited a need for an NSFS fire control system that facilitates a Multiple Rounds Simultaneous Impact (MRSI) capability. The idea behind MRSI is to coordinate individual weapon TOF such that multiple rounds impact one or more targets simultaneously. Theoretically, MRSI would degrade the effectiveness of enemy artillery tactics such as "shoot and scoot" that seek to reduce their vulnerability. MRSI has the support of many subject matter experts who espouse the benefits of massed or volume fire. In 1996, Lieutenant General Paul Van Riper documented the requirement for volume fire in *Naval Surface Fire Support Requirements for Operational Maneuver From The Sea.*

The third study incorporated advanced TTP into a four-model consortium, which included the Integrated Theater Engagement Model (ITEM), the "Enhanced Lanchester" model (ELAN), the Target Acquisition Fire Support Model (TAFSM), and the Army's ARTQUIK model. Code changes were made primarily to TAFSM, the Army's premier

fire support model. The study concluded that advanced TTP and "shooting smart" were critical to the reduction of ERGM quantity required to support a MEF. The results also demonstrated the significance of increased magazine sizes. When magazine capacities were limited, ships spent much of the engagement off line replenishing their ammunition.

In 1999 JHU/APL completed an analysis of alternatives study which examined "the relative effectiveness in a land attack role of a 155mm Trainable Advanced Gun System as compared to a 155mm Vertical Advanced Gun System." (Zimm, 1999) Once again, TAFSM, ELAN, and ARTQUIK models were linked. The study concluded that overall a 155mm Trainable gun outperformed a 155mm Vertical gun as well as a 5-inch/62-caliber gun. This conclusion is in agreement with the most recent recommendation made by United Defense, the prime contractor for the DD 21 gun design, for a traditional, turreted gun in lieu of a vertical gun. The Navy has concurred with this recommendation and will pursue a trainable gun solution. (Skibitski, 1999)

The debate continues over what land attack missiles to deploy on Aegis cruisers and destroyers to improve NSFS capabilities. For more than three years, the Navy has wrestled with this decision of what NSFS missiles to install on these surface combatants. In April 1999 Chief of Naval Operations Admiral Jay Johnson agreed with a recommendation for the Navy to purchase LASM. The Navy considers the procurement of LASM to be more cost-effective than converting the ATACMS to NTACMs. The Navy plans to convert 800 to 1,200 aged SM-2s to outfit 22 Aegis cruisers and 27 Aegis destroyers. (Schweizer, 1999) Meanwhile, NTACMS builder Lockheed Martin has begun an intense lobbying campaign, asserting that NTACMS will be less costly than LASM because, with a larger warhead and greater range, fewer missiles will be required

to destroy enemy targets. JHU/APL conducted the most recent evaluation of the two missiles, but neither missile dominated the other in the study.

NSFSSim was created as a first step toward the goal of providing military planners and analysts with a component-based simulation tool that can aid in the formulation of integrated NSFS gun and missile firing policies against mobile/relocatable targets. While not definitive, the simulation model is designed to operate on different platforms and to possess significant flexibility such that modifications can easily be made to increase the resolution or focus of the model. For example, instead of analyzing the NSFS problem, NSFSSim could be extended to examine defensive firing policies for surface combatants against anti-ship missiles (ASM). Another desirable feature of NSFSSim is that its user can quickly modify input parameters and immediately run simulations using a new data set (Fig. 1). Appendix A discusses the data structures and Java source code that make this possible.

Having provided a brief discussion of the challenges for NSFS and an overview of some studies that have been performed to address relevant NSFS issues, the next section describes the structure of this thesis.

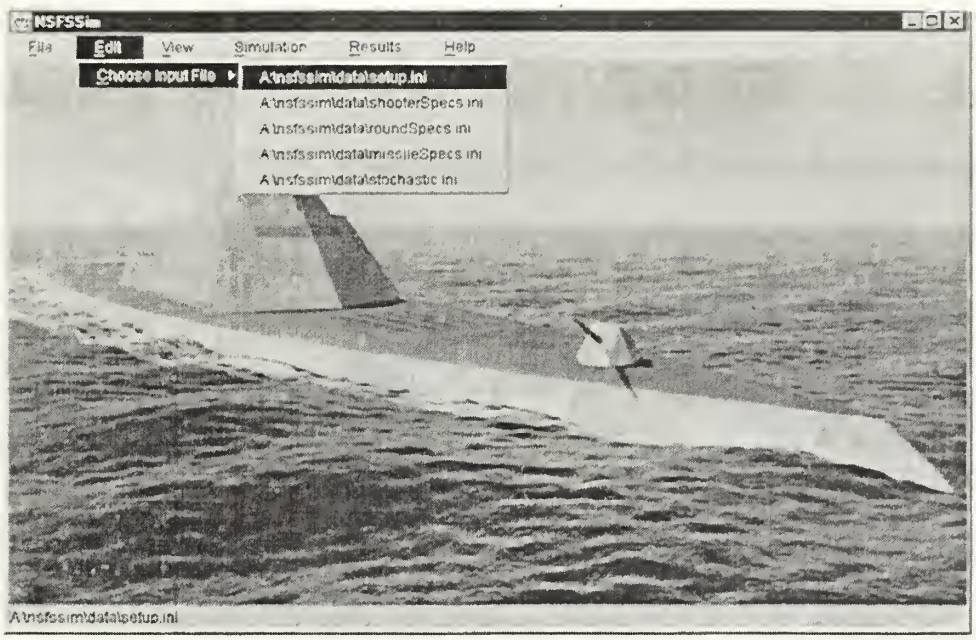

Figure 1. Editing Input Files in NSFSSim

NSFSSim allows the user to easily modify input data. By clicking on the Edit menu, the user can access any one of five editable data files. Changes to the data are made by modifying existing text fields and then overwriting the current file.

D. THESIS STRUCTURE

NSFSSim is a discrete-event simulation written using the Java programming language. As is the case with many simulation studies conducted at the Naval Postgraduate School (NPS), the flexible component architecture resident in NSFSSim is achieved by the use of Simkit, a discrete-event simulation package authored by Assistant Professor Arnold H. Buss and Lieutenant Kirk Stork, United States Navy (USN). (Stork, 1996)

The next chapter will provide a detailed description of this analysis tool, focusing on its development as well as the logic, assumptions, and interactions that drive the

model. Chapter III will offer an account of the types of analysis that can be conducted using NSFSSim. Finally, Chapter IV will summarize the results of the study, offering conclusions and recommendations for further research.

II. NSFSSIM

NSFSSim was developed as an analytical tool to provide insights into the problem of optimizing advanced NSFS weapons employment against mobile, short dwell time targets. The model's object-oriented design enables its extension to the fulfillment of other purposes beyond this application. Conceivably, NSFSSim could be used to address the following issues relevant to simulation studies (Townsend, 1999):

- Hardware acquisition, in which the new system (or additional purchases) are evaluated for their comparative worth.
- Force structuring, in which the force is shaped to incorporate the correct ratio of weapon systems of the right types.
- Tactical Development, in which non-lethal simulation can identify potential strengths and weaknesses of certain tactics.
- Capability of Forces, where the ability of the force to accomplish missions in theater is evaluated.

NSFSSim uses a discrete-event simulation methodology that is written in Java and uses some of Simkit's existing components and functions.

A. METHODOLOGY

The decision to utilize Java and Simkit to build NSFSSim was an easy one. Java offers platform independence, security, and powerful programming capabilities that are

not found in other languages. Simkit, which is written in Java, likewise provides a wealth of software components. When properly combined, or "loosely coupled," these components can produce a robust and flexible discrete-event simulation. (Bradley and Buss, 1998)

Simulation methodology was chosen to investigate NSFS firing sequences because of the intrinsic properties of the modern battlefield. Forces interacting on a modern battlefield will exhibit stochastic properties. Many interrelationships combine to create a complex, non-linear situation. A discrete-event simulation can model the dynamic processes associated with the modern battlefield. As is the case with most real-world systems, the NSFS problem is too complex to be evaluated analytically using a purely mathematical method. On the other hand, by virtue of today's powerful computers, a simulation enables a relatively rapid numerical evaluation of the problem.

Within NSFSSim, a discrete-event mechanism was used to advance the simulated clock. The state variables in a discrete-event simulation change instantaneously at certain points in simulated time, which correspond to the occurrence of events. Simkit provides all of the basic tools needed to construct a discrete-event simulation: a mechanism for scheduling events, updating an event list as events occur, and removing events from the event list.

Having presented the general methodology of NSFSSim, we next turn to a brief discussion of object-oriented programming (OOP) principles as a precursor to the more-detailed modeling aspects of NSFSSim.

B. MODELING PRINCIPLES

Before beginning an overview of NSFSSim's component-based design, it is useful first to provide a rudimentary introduction to OOP definitions and modeling principles. This section provides a brief description of OOP and its important design concepts, such as inheritance and encapsulation. In addition, unique Java modeling concepts will be presented.

1. Object-Oriented Programming

OOP has redefined the ways software developers think about and design their programs. Traditional, procedure-structured programming focuses on the design of algorithms and using data structures to manipulate those logic functions. OOP reverses this approach, focusing first on the design of the data structures and then incorporating functions into the data structures. "Simply stated, object-oriented design is a technique that focuses design on the data (= objects) and on the interfaces to it." (Hortsmann and Cornell, 1997)

A central concept in OOP is designing the data structures, or objects, such that each is responsible for executing a group of related tasks. When an object relies on functions or properties of another object, the former should "ask" the latter for the desired information via method calls rather than directly manipulate that object's data. In this manner, internal data and information remains hidden within objects. This principle of data hiding, referred to as *encapsulation*, enhances reusability and tends to minimize the time it takes to debug programming errors.

In OOP classes are templates for objects. The *class* is the single most important component in OOP design because it is the blueprint from which an object is actually constructed. When one creates an object using a class template, one is said to *instantiate*, or create an *instance* of, an object. For example, with a line of code like

```
SPArtillery artillery = new SPArtillery();
```

the `new` operator is used to create an artillery object (instance) of the `SPArtillery` class. In OOP terminology, the object is *instantiated.* In OOP each object generally consists of accessible functions, or *methods*, and data, or *instance variables.*

OOP allows one class to inherit the behavior, or methods and instance variables, of another. The motivation for this modeling principle, commonly called *inheritance*, includes reuse and abstracting common elements among classes. Other terms related to inheritance are *superclass*, *subclass*, and *extends*. The class from which another class inherits its functionality is called the superclass; the inheriting class is the subclass. Said another way, the subclass extends the superclass. The notion of extending a class is attractive because one is able to reuse the desirable behaviors of the superclass; at the same time, one is able to add or change behaviors to adapt to changing needs or for the purpose of specialization. To extend a class in Java, one uses the keyword `extends`. For example, the line

```
public class DD21 extends NSFSShip {
```

says that the `DD21` class inherits the behavior of the `NSFSShip` class.

Unlike some OOP languages, Java does not allow *multiple inheritance*. That is, a Java class can extend only one class. However, Java provides the notion of an *interface*, a powerful feature that affords the developer the ability to abstract common methods

from more than one class. The interface construct in effect replaces multiple inheritance of classes with multiple inheritance of interfaces. An interface, which contains no concrete methods or variables of its own, is essentially a contract signed by any class that *implements* it. The contract is to provide, or implement, every method in the interface. The implementing class is free to decide the internal workings of those methods. For example, NSFSSim uses a `Weapon` interface that consists of the following lines of code:

```
public interface Weapon {
   public double getMaxRange();
   public double getLethalRadius();
   public double getProbKill(Mover target);
}
```

The `NSFSWeapon` class implements the `Weapon` interface by using the keyword `implements`:

```
public class NSFSWeapon extends SimEntityBase implements Weapon {
```

This code promises that the `NSFSWeapon` class will have a `getMaxRange` method, a `getLethalRadius` method, and a `getProbKill` method that takes a `Mover` object. `Mover` itself is an interface implemented by the `BasicMover` class in Simkit.

2. The Listener Pattern

Java's interfaces can be used to implement a "listener pattern," another important modeling principle utilized extensively in Simkit. Implementing classes use the `Listener` interface for the purpose of handling events, specifically GUI events such as mouse clicks. The idea here is that a model's view should change in response to GUI events. The listener pattern enables an interested "listener" to be notified of events as they occur so that views may be modified accordingly. Java's event handling mechanism can be summarized in the following manner (Horstmann and Cornell, 1997):

- A listener object is an instance of a class that implements a special interface called (naturally enough) a *listener interface.*
- An event source is an object that can register listener objects and send them notifications when events occur. These notifications are methods of the listener interface.

A listener object is registered with the source object with the following general line of pseudo-code:

```
EventSourceObject.addEventListener(EventListenerObject);
```

Simkit applies the same event-notification pattern but emphasizes simulation events and object state changes. Simkit's listener pattern, likewise, is implemented with one line of code:

```
SimEventSource.addSimEventListener(SimEventListener);
```

In Simkit a `SimEventListener` object registered to a `SimEventSource` will be notified of each `SimEvent` (a Simkit method with the prefix "do") for which it has an identical event. Suppose, for example, that a `CounterBattery` object named radar is registered as a `SimEventListener` with an `ArtilleryBattery` object named battery. The code would look something like this:

```
battery.addSimEventListener(radar);
```

Now suppose that the ArtilleryBattery class has a "FireRound" event constructed as follows:

```
public void doFireRound() {
   …internal code for this method
}
```

If the radar instance wants to be notified of the battery's "FireRound" event, the `CounterBattery` class would have to have a method with exactly the same method construction—that is, a `public void doFireRound()` method, in which the internal

code may be different from that of the source method in the `CounterBattery` class. Simkit's implementation of the listener pattern enables efficient event handling within a simulation model with little more than a few lines of code.

3. Third Party Components

In addition to making extensive use of Simkit's `SimEventListener` pattern, NSFSSim borrows Simkit's notion of third party components. Simkit provides a non-partisan `Referee` class to adjudicate detections within a simulation. The Referee's tasks include maintaining a list of all targets and sensors and scheduling detections when a `Mover` or a `Sensor` starts moving. Like `Mover`, `Sensor` is an interface. Generally speaking, when the Referee determines that a target is within the range of a sensor, the Referee by default creates a `CookieCutterMediator` instance that implements the `Mediator` interface. In this manner, a mediator is created only when needed and is responsible for adjudicating the actual interactions between a single sensor and a single target.

Because movers and sensors should not be entrusted with the responsibility of determining their own detections, the referee and mediators are created as third party components to serve as honest brokers in the determination of sensor-target interactions. Although NSFSSim does not utilize Simkit's existing `Referee` and `CookieCutterMediator` classes, it applies the same modeling principles to build third party components to adjudicate the interactions between weapons and targets. These components will be discussed at the end of the chapter.

4. Manager Components

Whereas third party components are not allied with a particular side in a combat simulation, manager components within NSFSSim are created with the express purpose of directing the actions of a particular `Mover` implementation. The use of managers is a practical application of object-oriented or component-based design. In its most rudimentary form, a mover is responsible for executing movement events and reporting its implicit state within the discrete-event paradigm. A mover's manager serves in a command and control capacity to direct the mover to its next location and schedule other events that may be associated with the mover depending on its classification. For example, an `ArtilleryBatteryManager` instance directs its subject artillery battery to random locations on a two-dimensional battlefield and schedules "StartEmplacement," "EndEmplacement," "FireRound," "StartDisplacement," and "EndEmplacement" events for the artillery battery. From a design standpoint, using manager components to separate basic movement functions from other actions is desirable. Once again, this modeling concept serves to increase reusability and minimize debugging time.

Having provided a brief introduction to the modeling principles and terminology used in NSFSSim, we now turn to a description of the physical structure of NSFSSim and the actual classes used to build the simulation model.

C. NSFSSIM STRUCTURE

NSFSSim consists of a Java package named "nsfssim," an input data directory, an icon directory that contains graphical images to populate the model's views, a default

output directory, and a help directory. Each set of simulated engagements may either be viewed in the animation mode (Fig. 2), as a textual display of the event list, or in "silent" mode; Appendix B discusses the creation of animation in NSFSSim. Pertinent data is collected throughout and is written to a default text file in the output directory at the conclusion of each set of runs.

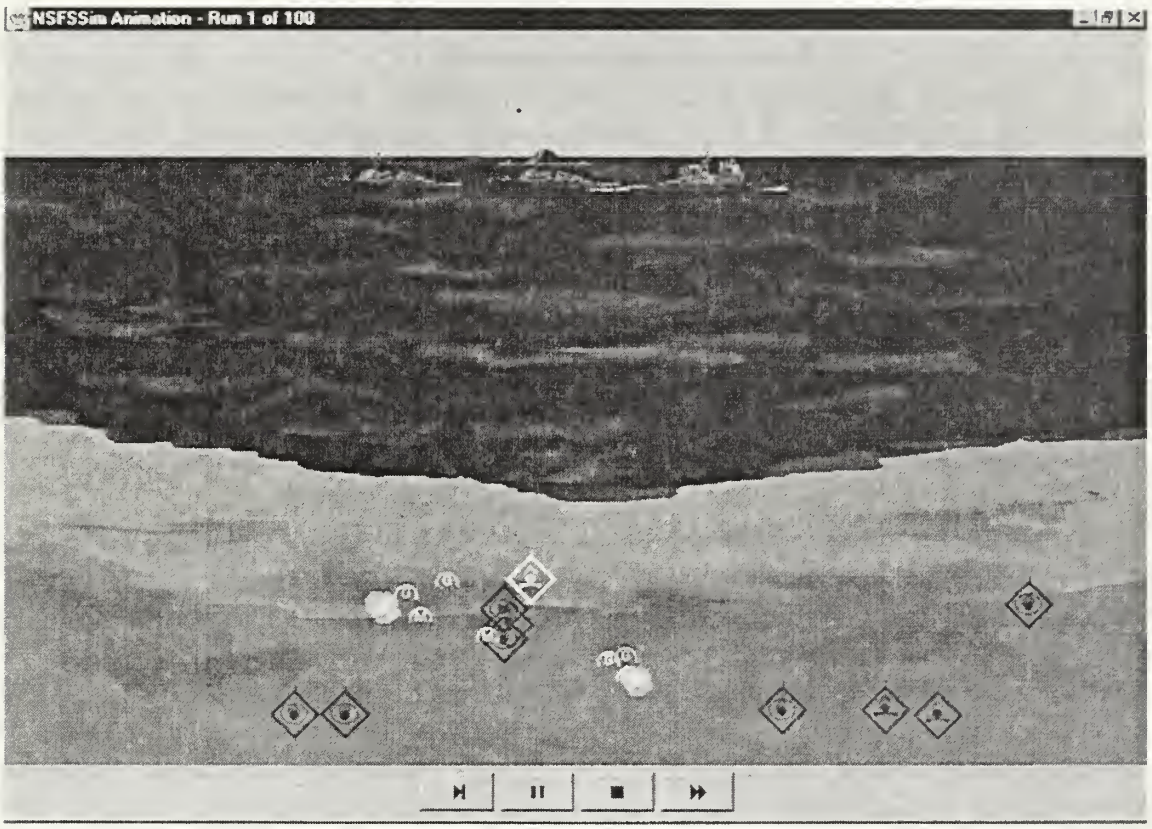

Figure 2. NSFSSim's Animation Mode

The animation mode provides a visual display of the running simulation. This screen shot shows two DDG 51's and one DD 21 on station conducting NSFS. A CG 47 cruiser is enroute to the ammunition onload rendezvous point to replenish its ammunition inventory. Artillery batteries are depicted in the foreground. Those rendered in red are at full strength. Any battery rendered in yellow is firing artillery rounds. Each gray battery has had one or more of its guns destroyed. An explosion indicates that at least one gun in a battery has just been destroyed. Once a battery has had all its guns destroyed, it is removed from the screen (left explosion). The white semi-circles depict ERGM (G) and LASM (M) fired from the surface combatants.

Because NSFS is conducted by surface combatants whose stand off ranges minimize their susceptibility to enemy counterattack, NSFSSim uses a predator-prey design. That is, the NSFS ships within the model are "predators" that use tactical fires to defeat enemy self-propelled and towed artillery batteries, the "prey." As the names imply, during the course of a simulated engagement, the ships are invulnerable, while the artillery batteries invariably are attrited.

Although real-world threats pose formidable challenges in the realm of simultaneity of missions, the thesis' singular scope of investigating favorable NSFS gun/missile firing policies obviated the need to model other mission areas. As such, no enemy surface combatants, aircraft, or submarines were modeled. Furthermore, Marines and Army troops, for which NSFS is designed to protect and empower, were omitted from the model.

NSFSSim models such entities as the NSFS surface combatants of the next decade, two of the advanced NSFS weapons that are being developed to advance 21st century warfighting concepts, and two types of enemy field artillery batteries. The movers exhibit simple linear motion and interact on a two-dimensional battlespace. These simplifying assumptions are made possible due to the fact that ERGM and LASM will use GPS assets only for precision guidance to each weapon's respective aim point. The weapons' lack of active radar seekers precluded the necessity of modeling the target acquisition process, which otherwise would have mandated the extension of the battlespace to a third dimension and would have introduced the problem of weapon-target geometry. The entities that will execute the firing policies that are being investigated in this thesis are `NSFSShip` instances.

D. NSFS SHIPS

Figure 3 illustrates the class hierarchy for NSFS ships in NSFSSim.

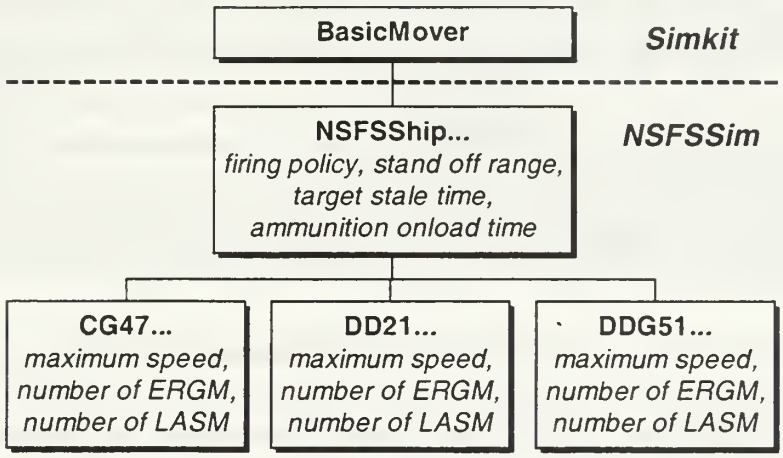

Figure 3. NSFSShip Component Hierarchy

The NSFSShip class extends Simkit's BasicMover class. CG47, DD21, and DDG51, in turn, subclass NSFSShip. The italicized text within each box indicates modifiable input parameters.

The `NSFSShip` class is the superclass for the surface combatants in NSFSSim. `NSFSShip` itself extends the `BasicMover` class in Simkit. Therefore, each NSFS ship inherits the behavior of a `BasicMover`. Specifically, each ship exhibits uniform linear motion. Additionally, `NSFSShip` entities share the following user-specified parameters: firing policy, stand off range, ammunition onload time, and target stale time (i.e., the maximum time a target aim point can reside in the engagement queue before it is deleted).

The software components used to model the surface combatants include the `DD21`, `DDG51`, and `CG47` classes, each extending `NSFSShip` (Fig. 3). While the names of the ship classes may appear to be confining, some flexibility is provided to uniquely

configure each ship type. Maximum ship speed as well as ERGM and LASM inventories may be specified for each ship class. The model's user may specify the creation of as many of each of the surface combatants as he desires.

For each DD21, DDG51, and CG47 that is created, NSFSSim instantiates a manager component, which is discussed in the following section.

E. NSFS SHIP MANAGERS

Each `NSFSShip` is controlled by an individual `ShipManager` instance. Based on the firing policy, the manager directs the execution of its designated ship's fire mission. The firing policy is an independent "variable" specified by the user prior to a set of runs. In NSFSSim, a firing policy consists of a sequence of characters, or a Java String—g's, G's, m's, M's, l's, and L's are the only accepted characters—where a "g" or a "G" represents a "ShootGun" event, an "m" or an "M" corresponds to a "ShootMissile" event, and an "l" or an "L" schedules a "Look," or kill assessment event. For example, to specify a Shoot (missile), Look, Shoot (missile) firing policy, one would enter either the String "mlm" or the String "MLM" in the NSFSShip firing policy field in NSFSSim's setup dialog (Fig. 4).

As long as the `NSFSShip` has sufficient numbers of ERGM and LASM remaining to fully execute the firing policy, its ship manager will cause it to conduct assigned NSFS missions. If, for instance, the promulgated firing policy was to fire three ERGM followed by launching two LASM—"GGGMM"—at a given aim point, the ShipManager would direct the firing of the specified sequence of rounds and missiles using the ship's available gun(s) and launcher(s). The user may specify the probability

distributions that underlie the `ShipManager` class' processing time and firing duration between shots. The default times are derived from Uniform(a, b) distributions.

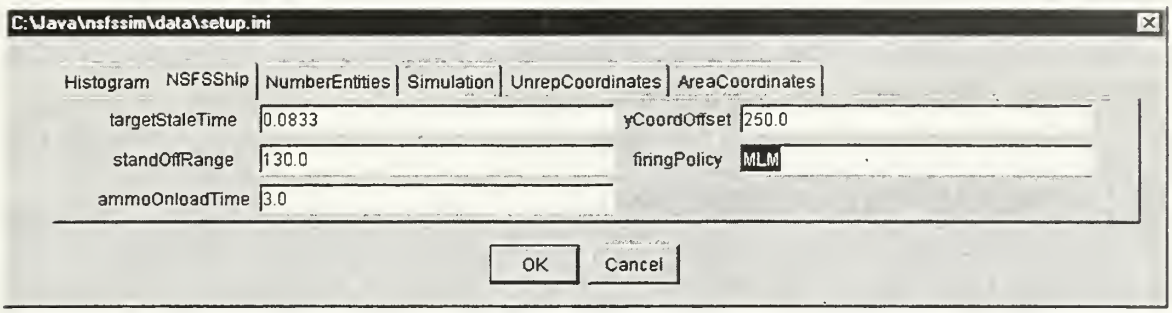

Figure 4. Specifying a Firing Policy

NSFSSim's setup dialog allows the user to modify parameters related to the histogram output, NSFSShip properties, number of model entities, simulation controls, and battlefield coordinates. This screen shot shows user-selection of the NSFSShip tab and the highlighting of the firing policy field. The String "MLM" indicates a Shoot (missile), Look, Shoot (missile) firing policy.

Once a ship's ERGM and/or LASM inventories are depleted below the level necessary to carry out the firing policy, the corresponding ship manager directs the ship to a user-specified ammunition onload rendezvous point. In actual combat conditions, a surface combatant likely would expend all its munitions and missiles prior to departing the operating area to replenish its ammunition. However, because this thesis only investigates the implications of specific firing sequences on enemy artillery battery effectiveness, NSFSSim in its present form disallows this eventuality. Future applications, on the other hand, could easily alter this behavior by extending the `ShipManager` class and rewriting a single method.

As would be the case in actual NSFS operations, the NSFSShip instance is unavailable for fire missions during the time it takes the ship to complete the ammunition

onload and return to station. Upon completion of the ammunition onload, the ship's ERGM and LASM inventories are reset to their initial levels. Back at its initial station, the ship once again is available to execute NSFS missions received from the mission-scheduling component, which will be discussed next.

F. NSFS MISSION SCHEDULING

Mission scheduling functionality resides within an instance of the `NSFSMission` class. The scheduler's logic in the present version is simple. From the set of ships that are on station and within maximum weapons release range of a mission aim point, the NSFSMission object randomly chooses a designated ship. This behavior can be altered easily to incorporate more complex shooter assignment and scheduling algorithms.

There are two major NSFS missions—counterbattery and call fire missions, both of which must be highly responsive in order to protect troops in contact and enable tactical maneuvers against the enemy. In the most general terms, a counterbattery mission is one that is initiated by countertargeting radar that detects the firing of enemy artillery rounds. Based on the trajectories of the artillery rounds, the radar system calculates an estimate of the firing gun's location. Engaged troops or forward observers (FO), on the other hand, generally initiate call fire missions.

NSFSSim creates two objects that generate these NSFS missions. The `CounterBattery` object is an instance of the `CounterBattery` class and determines the need for counterbattery missions. The `CallFire` instance is created from the `CallFire` class and generates call fire missions.

1. Counterbattery Missions

NSFSSim's `CounterBattery` object serves as a countertargeting radar. The `CounterBattery` class does not implement Simkit's `Mover` and `Sensor` interfaces; as such, the `CounterBattery` instance does not possess coordinate locations or a maximum sensor range. The `CounterBattery` instance relies solely on probabilities—specifically, the probability that its radar is on and the probability of detection—to determine the detection of individual artillery rounds.

Figure 5 depicts the logic flow for the generation of counterbattery missions. The `CounterBattery` instance listens to the "FireRound" event of each enemy artillery battery. As each round is fired, the `CounterBattery` object randomly checks for counterbattery detection. A detection occurs if two randomly drawn numbers are, respectively, less than the probability that the counter-targeting radar is active and the conditional probability of detection. Both of the probabilities are user-defined parameters. Given a detection, the `CounterBattery` instance generates a counterbattery mission against the subject battery. Target location error (TLE) is applied to the location of a randomly chosen gun within the battery to produce the mission aim point. The TLE distributions are x-coordinate and y-coordinate errors. By default, the distributions are Uniform(a, b), but this may be modified by the user.

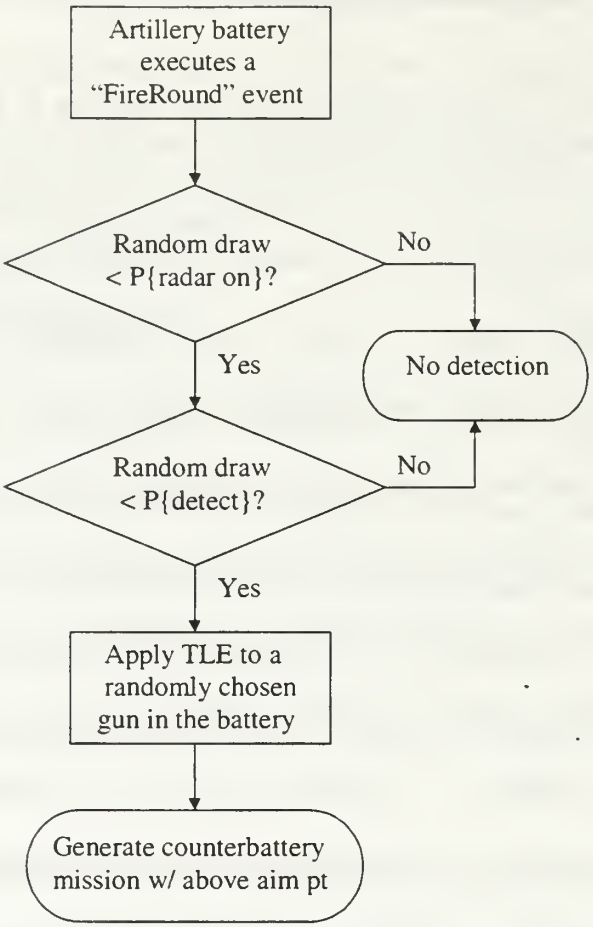

Figure 5. The Logic of Counterbattery Mission Generation

The random nature of NSFSSim's counterbattery mission generation is intended to simulate the uncertainty inherent in combat. Clearly, a counterbattery mission initiated at the beginning of a battery's firing sequence has a better probability of success than one that is queued by detecting the last artillery round fired.

2. Call Fire Missions

The `CallFire` object generates calls for fire through an arrival process. The user may change the probability distribution underlying the call for fire arrivals. By default the probability distribution is Exponential(λ) so that the calls arrive according to a

Poisson process. When a "CallForFireArrival" event occurs, the `CallFire` component randomly chooses one of the guns of an artillery battery and applies TLE to that location. At this point, the `CallFire` instance generates a call fire mission request with the calculated mission aim point. The x-coordinate and y-coordinate TLE distributions in the `CallFire` class are distinct from those in the `CounterBattery` class. As is the case in the `CounterBattery` class, the default TLE distributions are Uniform(a, b), but these too can be modified by the user.

Using the SimEventListener pattern, the requests for counterbattery and call fires are heard by the NSFSMission scheduler. Once the mission scheduler makes an assignment, the designated ship's manager is notified of the assignment and directs the execution of that mission. It is worth noting that, due to the randomness of the mission generation methodology, `NSFSShip` objects will often fire at a moving artillery battery, leading to the wasted expenditure of NSFS weapons.

The next section describes the component design of the targets of these NSFS missions—the enemy artillery batteries.

G. ARTILLERY BATTERIES

Figure 6 illustrates the `ArtilleryBattery` component hierarchy. The `ArtilleryBattery` class models enemy artillery batteries. Like the `NSFSShip` class, this class subclasses `BasicMover` and is also extended by other classes—`SPArtillery` and `TowedArtillery`. Each artillery battery is instantiated as a single mover. To model the battery characteristic, each instance has a vector of coordinates representing the individual gun locations within the battery.

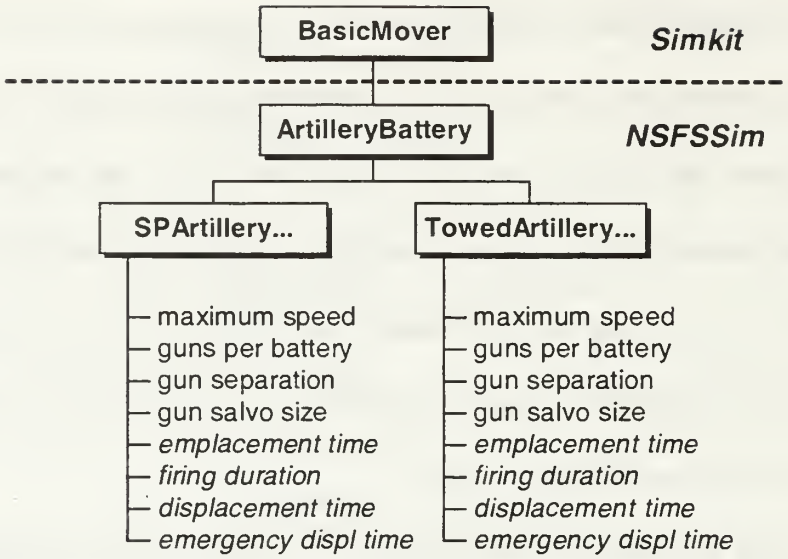

Figure 6. ArtilleryBattery Component Hierarchy

The SPArtillery and TowedArtillery classes extend the ArtilleryBattery class, which subclasses BasicMover. The user-defined data parameters are shown. Note that the italicized parameters represent probability distributions.

`SPArtillery` and `TowedArtillery` objects possess the same state variables. However, the specification of these variables is left to the user. Therefore, the characteristics of one battery type can be identical to that of the other, or as different as the user desires them to be. Realistically, the performance and vulnerability characteristics should be different. It is generally accepted that SPA gun systems are more capable and less vulnerable than towed guns.

Future SPA systems, such as the U.S. Army's developmental 155mm Crusader self-propelled howitzer XM2001, will possess state-of-the-art system survivability enhancement features. Most importantly, these weapon systems will possess increased mobility, speed, and firepower over today's field artillery systems. Automated rearmament—to include projectiles, charges, fuel, water, and lubricant—will increase

crew survivability by keeping the crew under armor, enabling continued availability for missions. (Foss, 1998)

Towed artillery systems, as the name implies, are less mobile than SPA systems. Reduced mobility on the battlefield equates to reduced survivability. Moreover, towed artillery guns do not enjoy the armored protection usually found on SPA weapons. Compounding this vulnerability is the higher manning level required to operate and maintain the towed systems.

In the execution of fire missions, however, SPA and towed artillery systems share common functionalities. NSFSSim structures artillery missions as a sequence of variable-time events:

- Movement to the geographic firing location
- Emplacement (i.e., preparations made prior to firing such as positioning spades and shooting azimuths)
- Firing of artillery rounds (the model assumes that each battery has an infinite ammunition inventory)
- Displacement (i.e., preparations made in advance of movement such as gun stowage for travel)
- "Scoot" (i.e., movement to a new location in avoidance of counterfire)

If, after emplacement, an artillery battery loses one or more of its guns to NSFS weapons fire, it immediately conducts a hastened, emergency displacement and scoots to a new location. The artillery battery in this case is deemed to be in distress and is unavailable to

conduct fire missions until it ends its "Scoot" event. The time it takes to conduct the emergency displacement as well as the above listed events are random times taken from probability distributions. Once again, the user may modify these distributions; by default, they are Uniform(a, b).

As is the case with the NSFS ships, each artillery battery is controlled by a manager component, which will be discussed next.

H. ARTILLERY BATTERY MANAGERS

The `ArtilleryBatteryManager` class provides the template for the creation of manager components that direct the missions of individual artillery batteries. To allow for future specialization, this class is subclassed by `SPArtilleryManager` and `TowedArtilleryManager`. Each manager instance is responsible for directing fire missions, as defined in the previous section. In controlling movement events, the managers choose uniform random locations on the two-dimensional battlefield, taking into account the stand off range of the surface combatants.

The actual firing sequence scheduled by a manager component depends on a number of factors. The number of "FireRound" events executed during an uninterrupted fire mission is determined by multiplying the number of surviving guns in the battery by the individual gun salvo size. The user may specify the salvo size, which by default is four rounds. The duration of the firing sequence is also dependent on the gun's rate of fire as defined by a firing duration probability distribution. This, too, can be modified by the user, the default being Uniform(a, b). The emplacement and displacement events that

are conducted, respectively, prior to and following the firing sequence are merely time delays placed on and removed from the event list.

Artillery battery fire missions are generated in a similar fashion to call fire missions. NSFSSim's user is expected to provide an arrival probability distribution for each of the two artillery battery types. The default mission arrival process for both battery types is the Poisson process. Fire mission generation and assignment are the responsibility of a single instance of the `EnemyMission` class. This object uses simple queuing theory to decide the assignments. When a towed artillery mission arrival occurs, for example, the `EnemyMission` instance assigns the mission to the `TowedArtillery` object with the smallest mission queue. Using the listener pattern, each `ArtilleryBatteryManager` object listens for these fire mission assignment events. The longer the artillery batteries remain in one location emplacing, firing rounds, and displacing, the bigger the window of opportunity for the NSFS weapons to achieve battery kills.

The next section discusses the components that model developmental NSFS precision munitions and missiles.

I. NSFS WEAPONS

The `NSFSWeaponMover` component hierarchy is depicted in Figure 7. The `Weapon` interface, introduced in Section B, provides methods for obtaining a weapon's maximum range, lethal radius, and probability of kill (PK) against a given `Mover` object. The `NSFSWeaponMover` class extends `BasicMover` and implements the `Weapon`

interface. The ERGMMover and LASMMover classes extend NSFSWeaponMover and represent developmental precision-guided NSFS munitions and missiles, respectively.

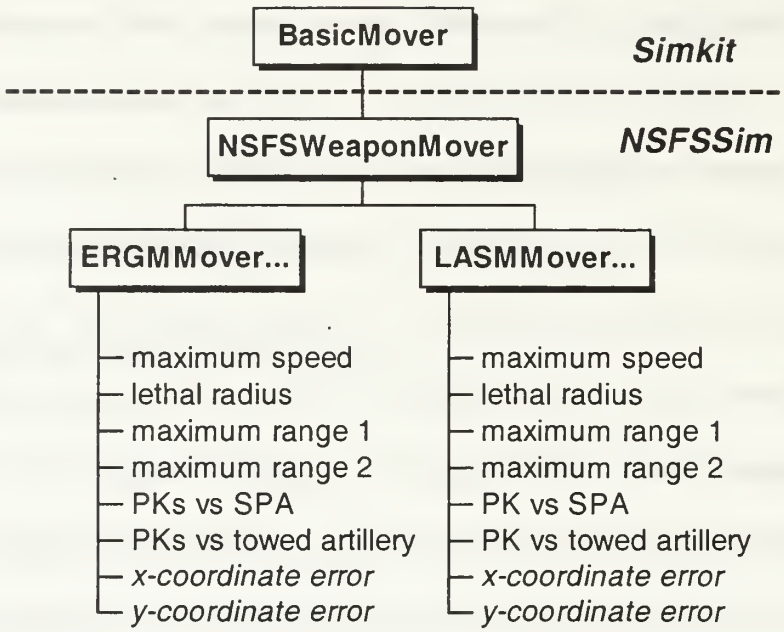

Figure 7. NSFSWeaponMover Component Hierarchy

The ERGMMover and LASMMover classes extend the NSFSWeaponMover class, which subclasses BasicMover. The user-defined data parameters are shown. The italicized parameters denote probability distributions.

An ERGMMover object is created each time a ship manager schedules a "ShootGun" event. Similarly, a "ShootMissile" event instantiates a LASMMover object. Each NSFSWeaponMover instance possesses the following modifiable parameters: maximum speed, lethal radius, maximum range 1 (used if the weapon is fired from a CG47 or DDG51 instance), maximum range 2 (in the case that the weapon is fired from a DD21 instance), x-coordinate error probability distribution, and y-coordinate error probability distribution.

ERGMMover and LASMMover instances also have user-specified, lethal radius-dependent PK values against SPArtillery and TowedArtillery instances. Because LASM is planned to contain a unitary warhead, the model allows the specification of only one lethal radius for the LASMMover class. The value of this lethal radius must match exactly the lethal radius specified to obtain the PK value against each artillery type. On the other hand, ERGM will feature a variable dispense diameter capability. The achievable dispense diameters will be 20 m, 40 m, 60 m, 80 m, and 100 m. Accordingly, NSFSSim provides the user the capability of specifying five different PK values for each of the two artillery battery types. Once again, for each battery type, the ERGMMover class' specified lethal radius must match one of the values required to obtain the PK values.

Each instantiation of an NSFSWeaponMover instance is accompanied by the creation of a weapon manager component. The NSFSWeaponManager class serves to control the flight of individual weapons and is subclassed by the ERGMManager and LASMManager classes. These managers are responsible only for applying weapon errors to the given aim point and then directing the specified weapon to the newly adjusted coordinates. At the end of a weapon's flight, a "WeaponImpact" event occurs. This event is heard by a third party component called the NSFSReferee.

J. NSFS REFEREE AND WEAPON TARGET MEDIATORS

The NSFSReferee object maintains registries of all the entities created at simulation run time. As NSFS ships, artillery batteries, and weapons change states, they may register or unregister with the referee. For example, when all of the guns in an

artillery battery are destroyed, the battery permanently unregisters, becoming unavailable to conduct fire missions or to be the target of NSFS missions. Similarly, when a `NSFSShip` instance departs to replenish its ammunition inventories, it unregisters with the `NSFSReferee` so that it becomes unavailable for NSFS mission assignments. The `NSFSShip` is added back to the ship registry when it returns to station.

In addition to performing this bookkeeping function, the referee works with a `WeaponTargetMediator` instance to adjudicate `NSFSWeaponMover` hits and misses. Each time the referee hears a "WeaponImpact" event, it directs an instance of the `WeaponTargetMediator` class to mediate the outcome of the weapon's impact.

The `WeaponTargetMediator` instance checks all of the gun locations of each registered ArtilleryBattery object. NSFSSim's `BatteryFormation` class is responsible for computing the actual gun locations. Given a user-specified gun separation value, each BatteryFormation instance positions an artillery battery's guns in a "Lazy W" pattern, the orientation of which is randomly decided at run time. As the name implies, the Lazy W pattern is a configuration in which the guns appear to form one or more linked "W's." This formation is commonly used to organize U.S. Army field artillery batteries, which nominally consist of six guns (Fig. 8). Figure 9 describes the logic flow for the adjudication of weapon-target interactions.

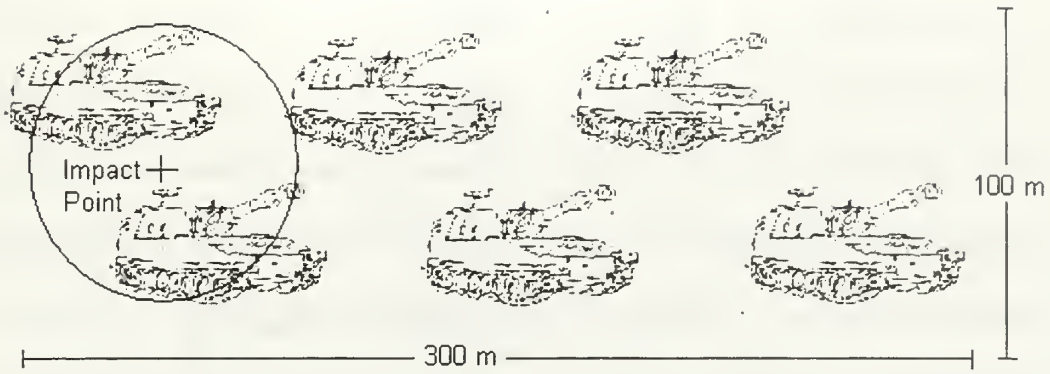

Figure 8. Typical "Lazy W" Battery Formation

This figure illustrates a typical battery formation consisting of six SPA guns dispersed over a 100 m x 300 m area. The circle shows the dispense diameter around the impact point of one ERGM round. In this case, the WeaponTargetMediator would conduct two independent, random draws to determine the kill assessments for the two guns located within the round's lethal area.

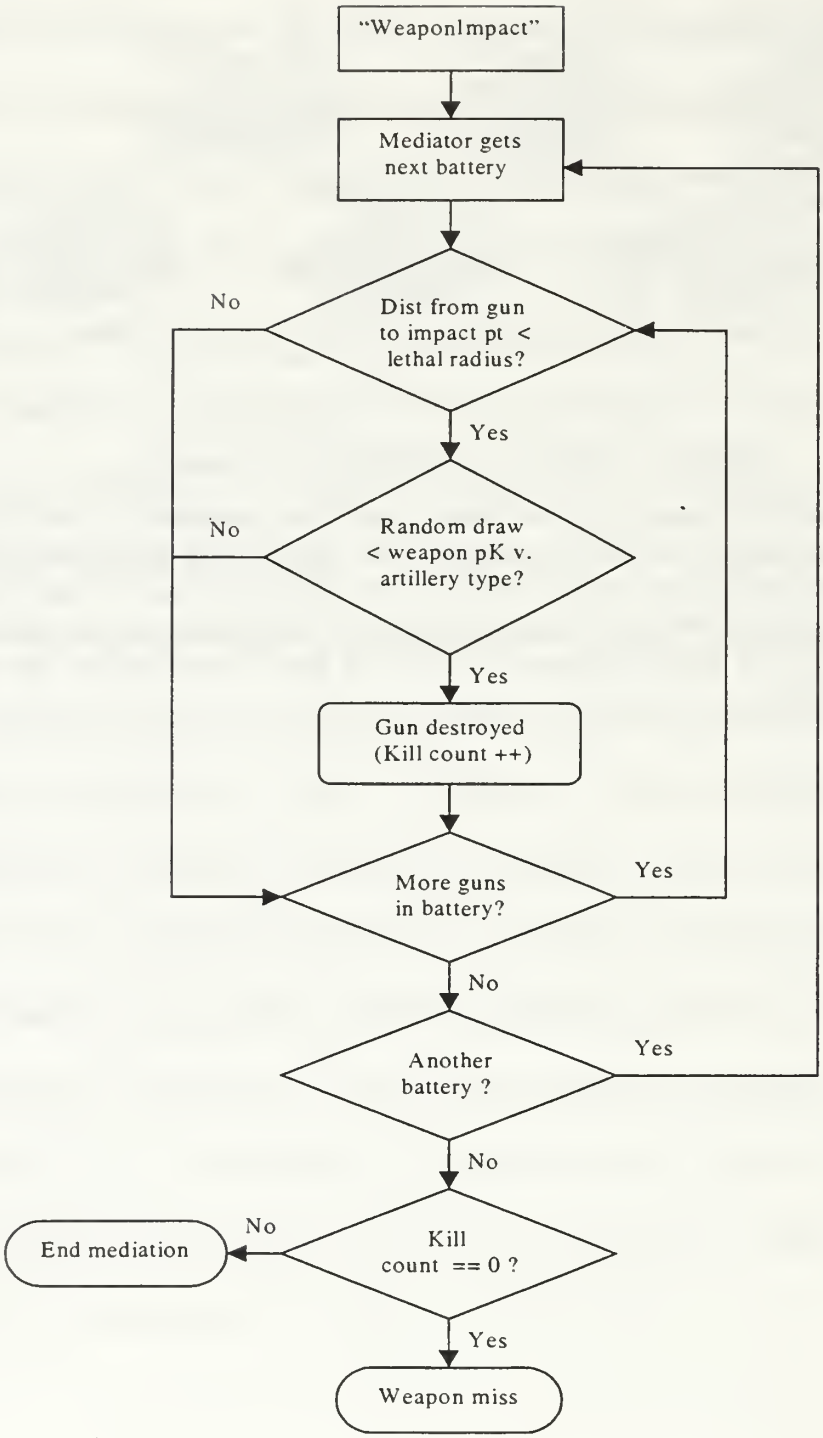

Figure 9. WeaponTargetMediator Logic

An artillery gun that lies within the destructive pattern of an NSFS weapon is destroyed with a certain probability of kill.

For those guns that lie within the lethal radius of the NSFSWeaponMover's impact point, the mediator conducts an independent random draw to determine hit or miss. If the random draw is less than the NSFSWeaponMover's PK value, for the specified lethal radius, against the particular artillery type, the mediator determines that a gun is destroyed. As mentioned previously, the determination of one of more gun kills while a battery is stationary causes the battery to conduct an emergency emplacement. The mediator performs this check for each registered artillery battery.

Having completed an overview of NSFSSim's simulation components, Chapter III will summarize the analysis using NSFSSim.

THIS PAGE INTENTIONALLY LEFT BLANK

III. ANALYSIS USING NSFSSIM

The previous chapters discussed the rationale behind the creation of NSFSSim as well as the modeling principles and components that provide the framework for the simulation model. This chapter summarizes the type of analysis that can be conducted using NSFSSim and briefly describes the use of the model. Recall that NSFSSim was constructed to analyze two specific problems:

1) Determine the best firing policy for ships conducting NSFS against mobile targets.

2) Determine the most effective ERGM dispense diameter against a given mix of artillery batteries.

Before describing the scenario used to analyze these issues, it is necessary first to discuss NSFSSim's relevant Measures of Performance (MOP) and the selection of an appropriate Measure of Effectiveness (MOE).

A. MEASURES OF PERFORMANCE

NSFSSim collects pertinent MOP data during each set of runs. These measures include:

- The average number of SPA missions conducted
- The average number of towed artillery missions conducted
- The average number of counterbattery missions conducted
- The average number of call fire missions conducted

- The average number of artillery rounds fired
- The average number of ERGM fired
- The average number of LASM launched
- The average number of SPA guns destroyed
- The average number of towed artillery guns destroyed
- The average number of SPA batteries destroyed
- The average number of towed artillery batteries destroyed

From this set of values, one can formulate several MOE alternatives to measure NSFS ship firing policy effectiveness against the artillery batteries. Obvious choices include analysis of alternatives (AOA), attrition-type measures such as the average number of total artillery guns destroyed divided by the number of NSFS weapons fired or, similarly, the average number of total artillery batteries destroyed divided by the number of NSFS weapons fired. Another MOE alternative is simply the average number of rounds fired by the enemy artillery batteries.

B. MEASURE OF EFFECTIVENESS SELECTION

The most appropriate MOE to measure the effects of firing policy changes on enemy artillery battery effectiveness appears to be the last alternative mentioned, the average number of rounds fired by the enemy artillery. An assumption in NSFSSim is that the destruction of at least one gun in a stationary artillery battery will cause the

battery to conduct an emergency displacement and move to a new location on the battlefield. All other events including the firing of artillery rounds are interrupted as a result of the emergency displacement. Therefore, an artillery battery that spends most of its time scooting to new locations will not fire as many rounds as one that remains mostly free from the harassment of effective NSFS weapon employment. Within this context, a truly effective firing policy is one that causes artillery batteries to scoot before they are able to fire their rounds. Having selected an appropriate MOE, the next section will describe the scenario used in the preliminary analysis using NSFSSim.

C. SCENARIO DESCRIPTION

The resolution of NSFSSim in its present state is limited, and actual data necessary to populate the model is either classified or unknown. As such, the scenario constructed to address the problems of firing NSFS weapons against mobile targets is a notional one using open source data. However, the scenario is reasonable for the purposes of this thesis and yields some useful insights.

The scenario is a seventeen-hour battle consisting of four surface combatants—two Aegis destroyers, one Aegis cruiser, and one 21st Century destroyer—conducting counterbattery and call fire missions against an even mix of six SPA batteries and six towed artillery batteries. The guns in each battery have a lateral separation of approximately 100 m, and each gun fires four rounds during an uninterrupted mission. The battlefield is a 230 nm by 65 nm rectangular region. The NSFS ships stand off 25 nm from the coast line, and the artillery batteries maneuver no closer than 15 nm to the coast. Therefore, LASM and ERGM, with speeds of mach 2.0 and mach 0.9,

respectively, must travel at least 40 nm to their aim points. The ship magazines are loaded such that, during the course of the seventeen-hour scenario, each ship likely will go off line to replenish its ammunition inventory at least once or twice. Each ammunition onload event lasts three hours, and it is assumed that there is no upper bound on the number of onload events that can occur simultaneously. The onload rendezvous point is located such that traversal times to and from the point equate to an additional hour of off station time for the replenishing ships.

SPA battery missions arrive more frequently than towed artillery missions. While both artillery battery types have a speed of 45 kph, SPA batteries on average have shorter dwell times than do the towed artillery batteries. In addition, by virtue of the NSFS weapon PK values against the artillery types, the SPA batteries are less vulnerable to destruction than are the towed types.

The next sections discuss some preliminary analysis using the NSFSSim model.

D. FIRING POLICY INVESTIGATION

A set of 100 runs without NSFS missions was conducted to establish a baseline measure of average rounds fired when artillery batteries are not targeted. Figure 10 shows the corresponding NSFSSim histogram for the number of artillery rounds fired. Figure 11 is a screen shot of NSFSSim's text editor showing the pertinent summary statistics of the baseline scenario. When no NSFS weapons are fired at the artillery batteries, the batteries fire on average 1355 rounds during a seventeen-hour period.

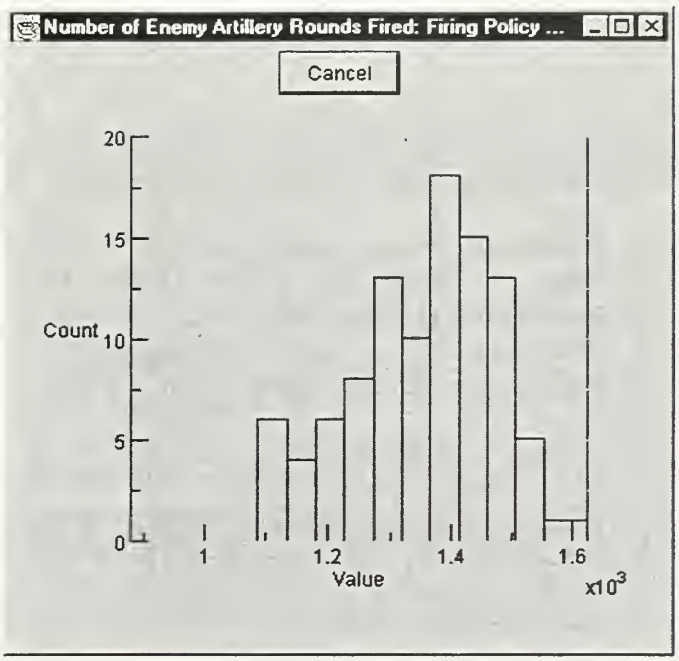

Figure 10. Baseline Histogram of Artillery Rounds Fired

If the histogram option is enabled in NSFSSim, the user is prompted to specify one of seven statistics to be graphed in a histogram that can be displayed at the end of a set of runs. This screen shot shows a histogram of the number of artillery rounds fired during 100 runs in which no NSFS weapons are fired.

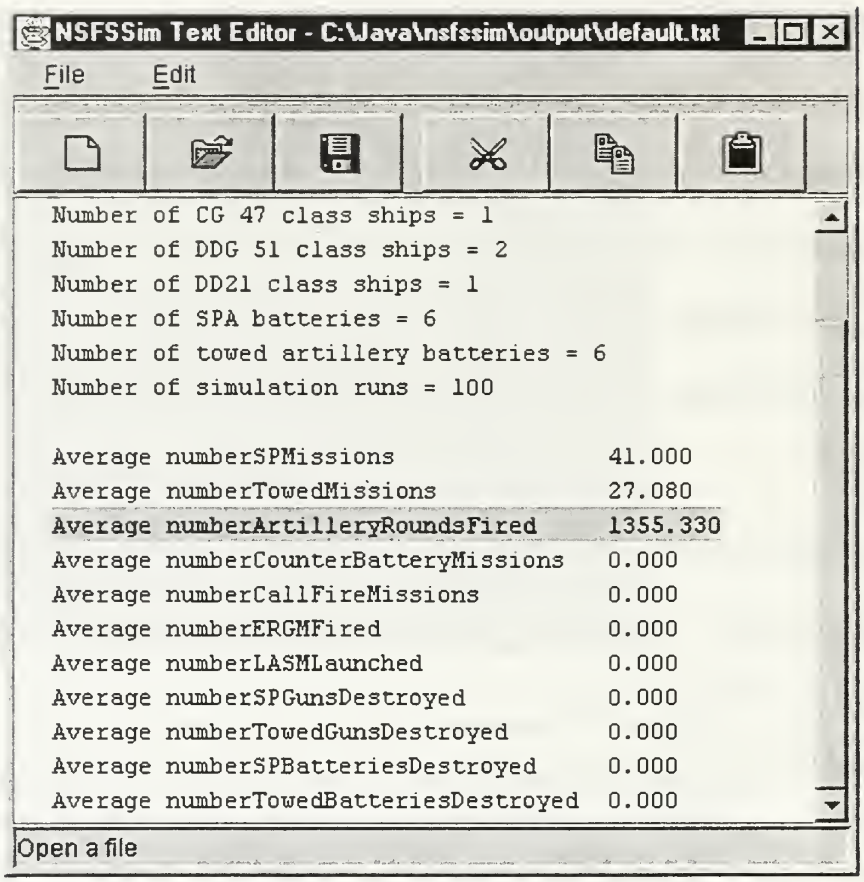

Figure 11. NSFSSim's Text Editor

NSFSSim's text editor can open, modify, and save any text file. At the conclusion of a set of runs, the user can open the default output file and read the pertinent data for the completed runs. This screen shot shows the default file for the baseline scenario in which no NSFS weapons are fired. Note that on average 41 SPA missions and 27 towed artillery missions are conducted when the artillery batteries are unharassed.

The first firing policy tested was a sequence of five ERGM followed by three LASM, or "GGGGGMMM." The ERGM dispense diameter was set at 60 m. This scenario was performed 100 times. Figures 12 and 13 show the text editor frame and histogram frame, respectively, for this particular scenario.

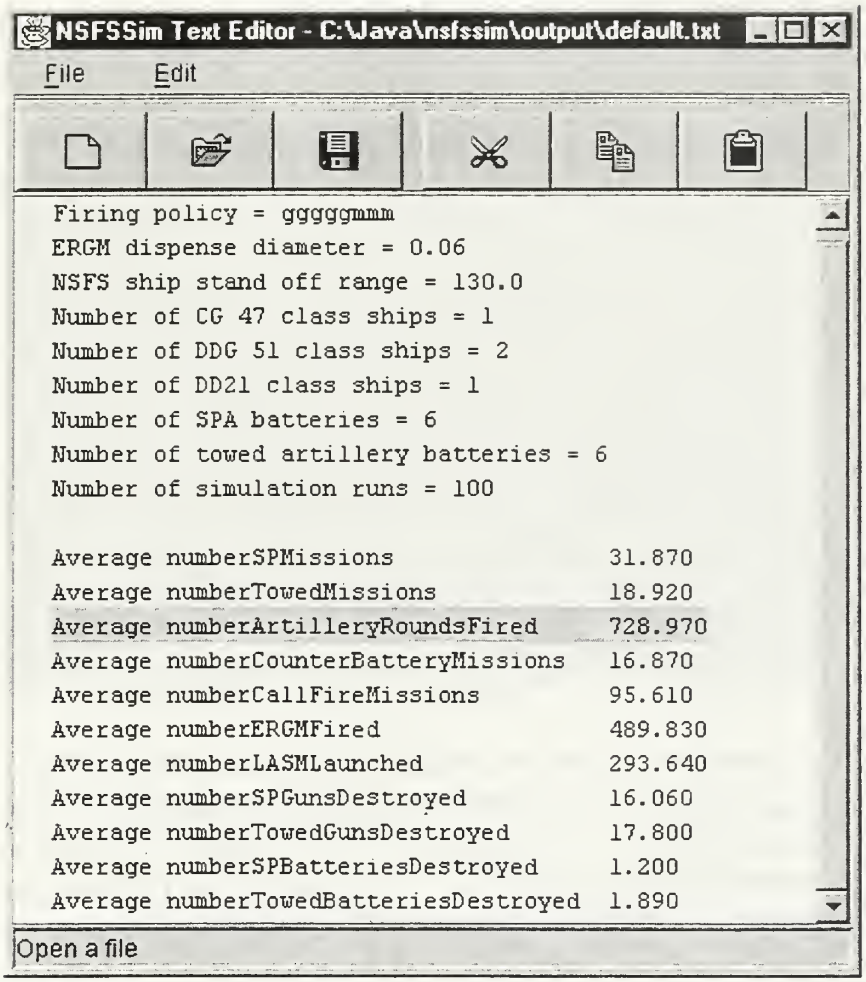

Figure 12. Firing Policy GGGGGMMM

During each counterbattery or call fire mission, each surface combatant fired five ERGM and launched three LASM in executing this firing policy. The times between the weapon firings were drawn from the specified firing duration distribution. With this firing policy the average number of artillery rounds fired was reduced from the baseline level to approximately 729 rounds during each seventeen-hour battle.

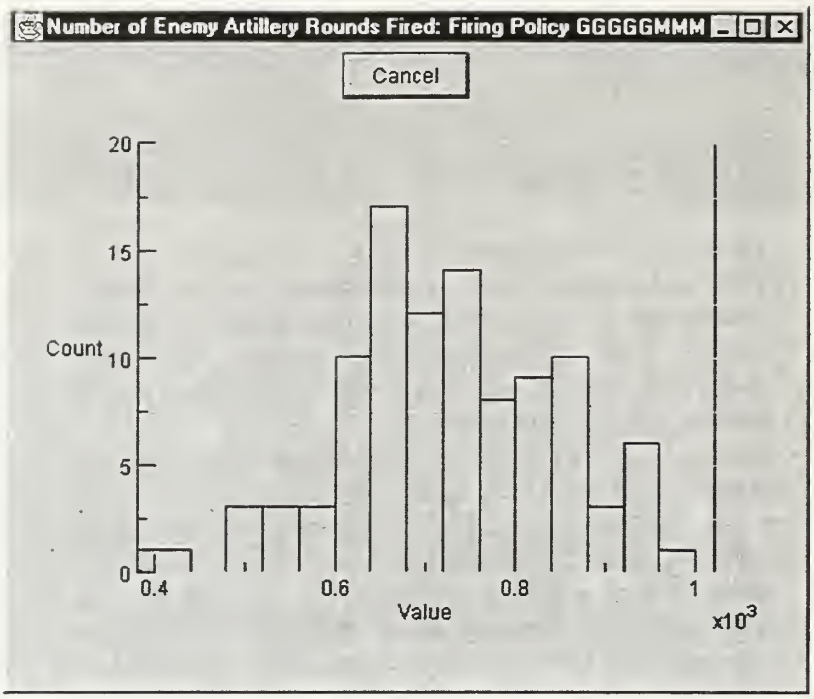

Figure 13. Histogram for GGGGGMMM Firing Policy

This screen shot shows the corresponding histogram of the number of artillery rounds fired when the NSFS ships used a GGGGGMMM firing policy. The mean value obtained by performing 100 runs was approximately 729 rounds.

The next firing policy examined was also a sequence of five ERGM and three LASM. This time, however, the missiles were launched prior to the firing of the precision rounds—a MMMGGGGG firing policy—to determine whether or not the order of the weapons fired has an effect on the effectiveness of the enemy artillery batteries. This scenario was performed for a set of 100 trials. With this firing policy, the average number of artillery rounds fired was approximately 692 rounds. Figures 14 and 15, respectively, show the default output file and histogram for this firing policy implementation.

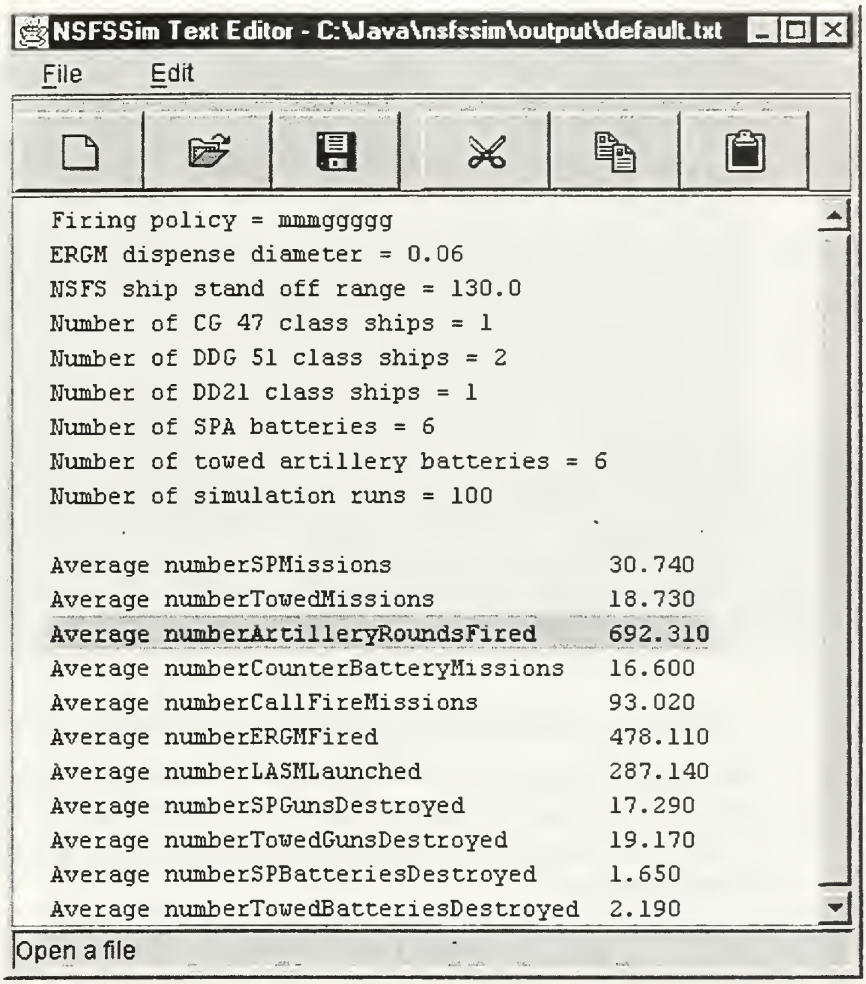

Figure 14. Firing Policy MMMGGGGG

By launching the missiles first, the NSFS ships improved upon the MOE that was achieved by firing ERGM rounds first. The number of artillery rounds fired during each run using the MMMGGGGG firing policy on average was approximately 692 rounds.

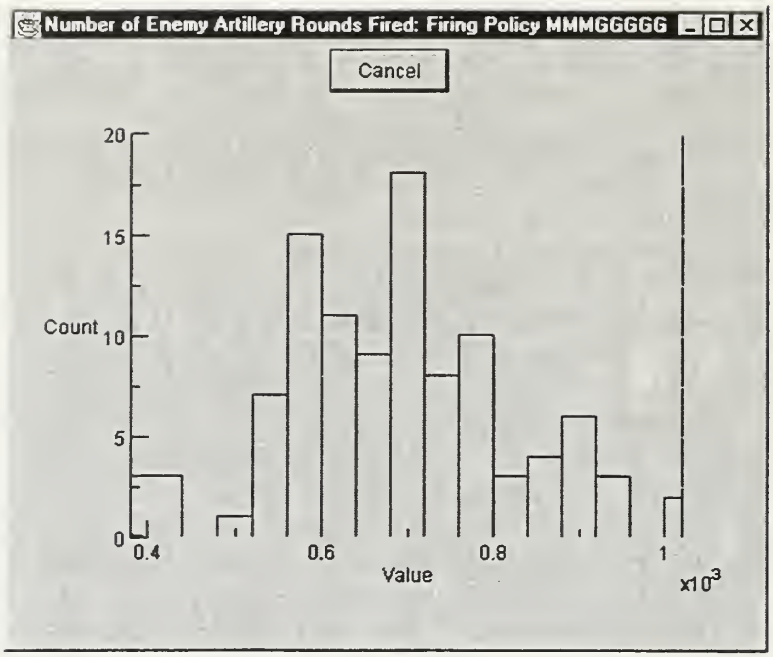

Figure 15. Histogram for MMMGGGGG Firing Policy

For this firing policy, the mean number of artillery rounds fired during each simulation run was 692.31 rounds. As before, the histogram was produced by conducting 100 simulation runs.

From the previous observations, it appeared that launching LASM, which is faster and more lethal than ERGM, earlier in the firing sequence decreased the effectiveness of the artillery batteries. Further testing with different numbers of missiles and munitions but the same structure yielded similar results. In order to determine if the differences in the mean number of rounds fired using the GGGGGMMM and the MMMGGGGG firing policies is statistically significant, a two-sample t-test was performed. The t-test was deemed appropriate because of the relatively large sample sizes involved and because the histograms showed comparable sample variances. The t-test produced a p-value of 0.0276. Therefore, at a significance level of 0.05, it was concluded that the true mean values are significantly different. This result was not surprising because the

responsiveness of NSFS weapons is tremendously important to the success of NSFS missions.

Somewhat surprising were the results for a MMGGGG firing policy. That is, two LASM and four ERGM per NSFS mission on average resulted in the firing of fewer artillery rounds by the enemy batteries. The output from 100 runs using this firing policy are shown in Figures 16 and 17.

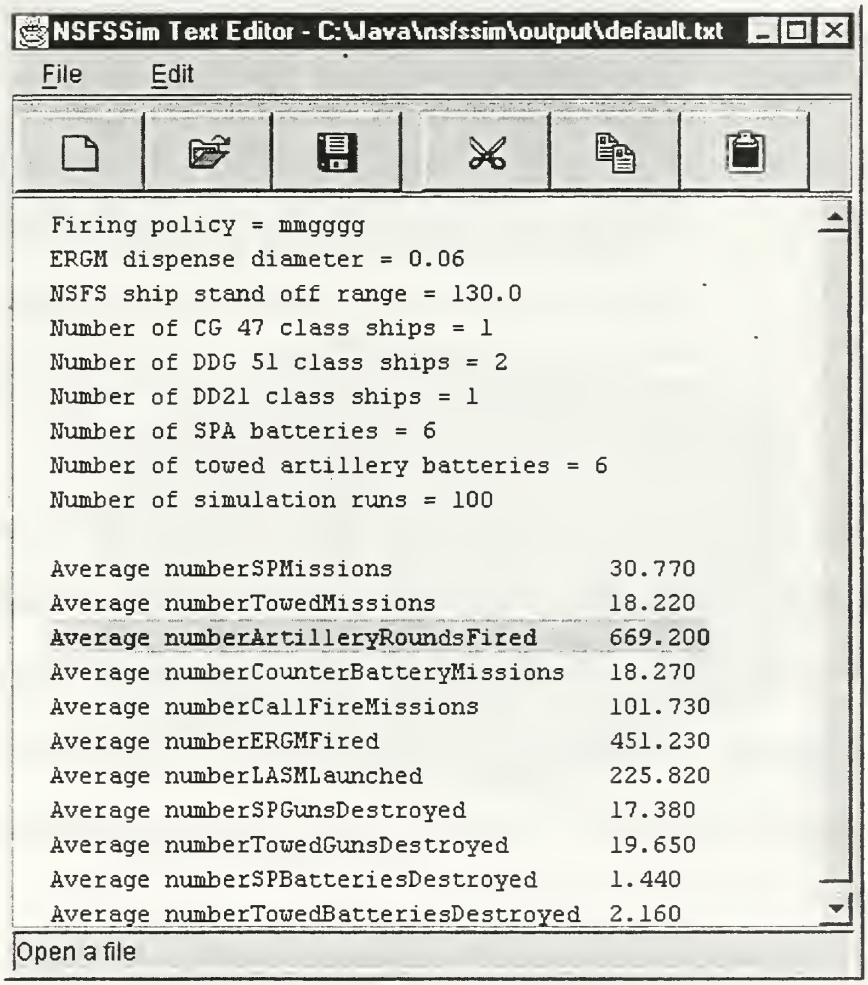

Figure 16. Firing Policy MMGGGG

This firing policy on average resulted in the artillery batteries' firing of approximately 669 artillery rounds per simulation run. Therefore, the selected MOE showed an improvement when the ships used six weapons instead of eight weapons as in the MMMGGGGG firing policy.

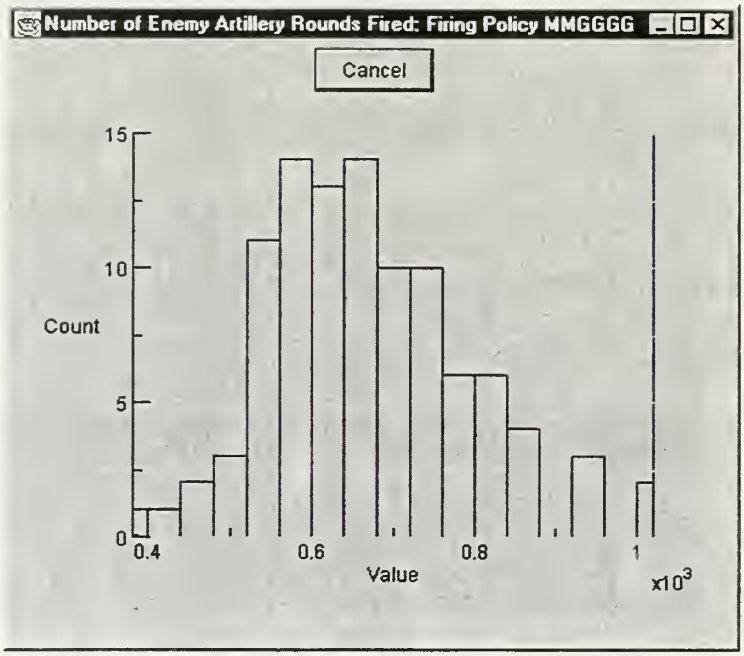

Figure 17. Histogram for MMGGGG Firing Policy

Repeated for 100 simulation runs, the MMGGGG firing policy resulted in an average of 669.20 artillery rounds fired by the artillery batteries during each battle.

The improvement in the MOE using fewer weapons may be attributable to the fact that firing more weapons each mission will force the surface combatants to depart their stations earlier and more often to take on more ammunition. This is a likely causal factor because the seventeen-hour scenario specifies an offstation time of four hours, during which time the replenishing ship is unavailable for counterbattery and call fire missions.

The next section presents a brief synopsis of preliminary analysis using different ERGM dispense diameters.

E. VARYING ERGM DISPENSE DIAMETER

The next question addressed in the study was whether an optimal ERGM dispense diameter existed for the given mix of artillery batteries in the scenario. In this analysis the MMMGGGGG and MMGGGG firing policies were used once again. For each set of 100 runs implementing one of the two specified firing policies, the ERGM dispense diameter was varied from 20 to 100 m in increments of 20 m (runs using a 60 m dispense diameter were completed for the previous analysis). Figure 18 shows the results of the ten sets of runs.

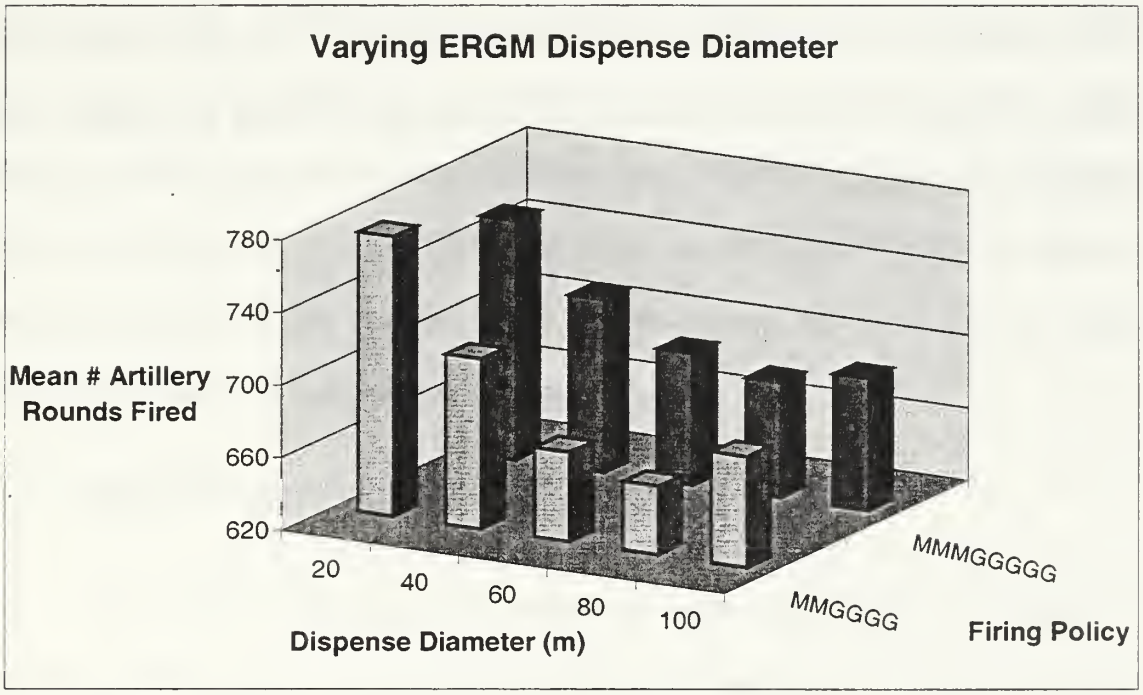

Figure 18. Effect of Varying ERGM Dispense Diameter

The plot shows that for this particular scenario an ERGM dispense diameter of 80 m on average resulted in the best results for MMGGGG and MMMGGGGG firing policies. That is, the enemy artillery batteries on average fired fewer rounds when the surface combatants used an ERGM dispense diameter of 80 m.

This analysis revealed that, despite higher associated PK values, small dispense diameters produced unfavorable results against dispersed mobile targets. Both a 60 m and 80 m ERGM dispense diameter resulted in marked improvements in the MOE, with the latter diameter fairing slightly better. A 100 m dispense diameter, although providing a larger lethal area, tended to be less effective due to smaller associated PK values. Different firing policy implementations yielded similar results.

This type of analysis could be extended easily to incorporate different mixes of SPA and towed artillery batteries or to include sensitivity analysis using varying battery characteristics. Clearly, the analysis conducted in this thesis is only preliminary and further analysis with more detailed models using "real" data is required. However, the analysis revealed some useful insights and emphasized the need for further model development and research.

IV. CONCLUSIONS AND RECOMMENDATIONS

A. THE NEED FOR ANALYSIS

This thesis has demonstrated the need for continued analysis in the area of tactical employment of advanced NSFS weapons. While the development of such sophisticated weapons as ERGM and LASM has proceeded with the momentum of adequate funding, weapons systems integration and tactical considerations remain at the conceptual stages. Compared to the relatively low cost of today's 5 inch rounds, these advanced weapons will be quite costly. Therefore, efficient utilization of these weapons in support of NSFS is an important issue. Moreover, the emergence of more capable, more mobile enemy weapons systems demands that these NSFS weapons be optimally employed. The Naval Surface Fire Support Simulation (NSFSSim) model has yielded some useful insights into the question of NSFS gun and missile employment against mobile targets, but further analysis using more complex models is required.

B. DEVELOPMENT OF NSFSSIM

Some studies addressing NSFS issues have used successfully a consortium of combat models to capture the complexities of the Joint land battle. However, due to the rigid design of these simulation models, major modification to existing code generally is required to enable the models to work together. This thesis pursues a one model approach, creating a new simulation model (NSFSSim) that features substantial flexibility such that it can operate on any hardware platform, can be extended easily to

provide greater resolution, and can be modified readily for future applications. NSFSSim is an analysis tool that allows the user to make changes to input parameters and run simulations without the burden of rewriting and recompiling any source code. In its present form, NSFSSim provides methods for analyzing the effectiveness of different firing sequences as well as the effectiveness of various ERGM dispense diameters.

C. RECOMMENDATIONS FOR FURTHER ANALYSIS

Preliminary analysis using NSFSSim reaffirms the importance of the responsiveness, range, and lethality of NSFS weapons employed against mobile, short dwell time targets. In particular, launching LASM early in a firing sequence is on average a better policy than launching the missiles after the ERGM rounds are fired. Additionally, setting the ERGM dispense diameter at 60 m or 80 m generally produces the best results against dispersed, mobile units. To be sure, these preliminary findings should be tested against other scenarios. In its present form, NSFSSim could be used to investigate the effectiveness of MRSI tactics against artillery batteries. By setting the firing duration to 0.0 and using the same speed for all of the weapons, all of the weapons fired for a particular NSFS mission would impact at the same time.

Further analysis with a more complex version of NSFSSim and "real" data is necessary to gain more insights into the problem of NSFS gun/missile firing policy. NSFSSim should be extended to use entities that exhibit more realistic movement. Sensors could be modeled as actual entities. Enemy artillery batteries could be modeled with the capability of utilizing countermeasures and decoy tactics. Furthermore, the

challenges facing surface combatants in the littorals should be modeled. Some of these challenges include land-based aircraft, land and sea-based missile systems, and mines.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: DATA STRUCTURES IN NSFSSIM

All of the input data in NSFSSim is stored within "ini" files in the data directory. The structure of an "ini" file lends itself to the creation of a data base in the form of a Hashtable of Hashtables, or a `Hashtable2` instance. The file "setup.ini" used in NSFSSim shows the typical structure of an "ini" file:

```
[NSFSShip]
firingPolicy = mmgggg
standOffRange = 130.0
yCoordOffset = 250.0
targetStaleTime = 0.0833
ammoOnloadTime = 3.0

[NumberEntities]
nsfssim.CG47 = 1
nsfssim.DD21 = 1
nsfssim.DDG51 = 2
nsfssim.SPArtillery = 6
nsfssim.TowedArtillery = 6

[AreaCoordinates]
lowerLeftX = 0.0
lowerLeftY = 0.0
upperRightX = 750.0
upperRightY = 480.0

[UnrepCoordinates]
xCoord = 418.0
yCoord = 250.0

[Simulation]
numberOfRuns = 100
stopTime = 17.0
stopWhenTgtsDead = false
singleStep = false
verbose = false

[Histogram]
leftValue = 400.0
rightValue = 1000.0
numberOfCells = 15
```

As illustrated by "setup.ini," a block of related data is specified by [ ]. Within each block, there exists any number of key-value pairs (e.g., firingPolicy = mmgggg). The `Hashtable2` class converts an "ini" file into a Hashtable of Hashtables so that one can access the value of a particular key-value pair by specifying the block and the key. NSFSSim provides a GUI that allows the user to modify and save the values within five "ini" files. `Hashtable2` and the classes that build this GUI are listed here:

```
// The Hashtable2 class

package nsfssim;

import java.util.*;
import java.io.*;
import java.net.*;

public class Hashtable2 extends Properties {

// constructors
   public Hashtable2() {
      super();
   }

   public Hashtable2(Properties prop) {
      super(prop);
   }

   public Hashtable2(URL url) {
      super();
      this.load(url);
   }

   public Hashtable2(File file) {
      super();
      this.load(file);
   }

// instance methods
   public void put(Object firstKey, Object secondKey, Object value) {
      Hashtable values;
      if (this.containsKey(firstKey)) {
         values = (Hashtable) this.get(firstKey);
      }
      else {
         values = new Hashtable(10);
         this.put(firstKey, values);
```

```
    }
    values.put(secondKey, value);
}

public Object get(Object firstKey, Object secondKey) {
    Hashtable values;
    Object returnValue = null;
    if (this.containsKey(firstKey)) {
        values = (Hashtable) this.get(firstKey);
        returnValue = values.get(secondKey);
    }
    return returnValue;
}

public void load(String fileName) {
    File file = new File(fileName);
    if (file.exists() ) {
        this.load(file);
    }
    else {
        throw new IllegalArgumentException("File " + fileName + " not
          found.");
    }
}

public void load(URL file) {
    this.load(new File(file.getFile()));
}

public void load(File file) {
    int lineNumber = 0;
    try {
        FileReader instream = new FileReader(file);
        BufferedReader input = new BufferedReader(instream);

        StringTokenizer tokens = null;
        Properties currentBlock = new Properties();
        String currentBlockName = "";
        for (String nextLn = input.readLine(); nextLn != null; nextLn
          = input.readLine()) {
            lineNumber++;
            if (nextLn.startsWith(";") || nextLn.startsWith("#")) { }
            else if (nextLn.startsWith("[") && nextLn.endsWith("]")) {
                tokens = new StringTokenizer(nextLn, "[]");
                if (tokens.countTokens() == 1) {
                    currentBlockName = tokens.nextToken();
                    currentBlock = new Properties();
                    this.put(currentBlockName, currentBlock);
                }
                else {
                    throw new RuntimeException(" on line " + lineNumber
                      + ":\n" + nextLn + "[# tokens = " +
                        tokens.countTokens() + "]");
                }
            }
            else {
```

```
                tokens = new StringTokenizer(nextLn, "=");

                switch (tokens.countTokens()) {
                   case 0:
                      break;
                   case 1:
                      currentBlock.put(tokens.nextToken().trim(), "");
                      break;
                   case 2:
                      currentBlock.put(tokens.nextToken().trim(),
                        tokens.nextToken().trim());
                      break;
                   default:
                      throw new RuntimeException (
                        "Improper format in " + file + " on line " +
                        lineNumber +":\n" + nextLn + "[# tokens = " +
                        tokens.countTokens() + "]");
                }
             }
          }
          input.close();
       }
       catch (FileNotFoundException e) {System.err.println(e);
         e.printStackTrace(System.err);}
       catch (IOException e) {System.err.println(e);
         e.printStackTrace(System.err);}
    }

    public Object put(Object key, Object value) {
       if (value instanceof Map) {
          return super.put(key, value);
       }
       else {
          throw new IllegalArgumentException("Hashtable2 can only accept
            Maps as values.");
       }
    }

}
```

```
// The INIFileEditor class

package nsfssim;

/**
  *  This class edits an INI file.
**/

import java.io.*;
import java.util.*;
import javax.swing.*;

public class INIFileEditor {
```

```
// instance variable
   private File file;
   private JFrame frame;

// constructor
   public INIFileEditor(File fileName, JFrame f) {
      file = fileName;
      frame = f;
      this.editFile(file);
   }

// instance method
   public void editFile(File file) {
      INIFileReader reader = new INIFileReader(file);
      JTextField[] fields = reader.getValueFields();
      JPanelDialog d = new JPanelDialog(frame, file.toString(), true,
        reader, fields, null);
      d.show();
      if (d.getValue() != null) {
         StringTokenizer tokens = new StringTokenizer(d.getValue());
         if (tokens.countTokens() == ((String[])
           reader.getValueNames()).length) {
            String[] values = new String[tokens.countTokens()];
            int k = 0;
            while (tokens.hasMoreTokens()) {
               values[k] = tokens.nextToken().trim();
               k++;
            }
            new INIFileWriter(reader.getFileName(),
              reader.getTabNames(), reader.getSubCounter(),
              reader.getValueNames(), values);
         }
      }
      d.dispose();
      return;
   }

}
```

// The INIFileReader class

```
package nsfssim;

import simkit.util.*;

import java.io.*;
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class INIFileReader extends JPanel {
```

```
   private INIFileProperties fileINI;
   private String fileName;
   private JTabbedPane pane;
   private JPanel[] tabs;
   private String[] tabNames;
   private String[] valueNames;
   private String[] values;
   private JTextField[] valueFields;
   private int counter;
   private Integer[] subCounter;
   private int subTotal;
   private int valueCounter;

/**
 * Creates an INIFileReader that sorts the keys of the INI file
 * into tabbed Panes and JLabels and allows the values to be
 * changed.
**/
   public INIFileReader(File file) {
      counter = 0;
      subTotal = 0;
      valueCounter = 0;
      pane = new JTabbedPane();
      Properties prop = null;
      fileName = file.toString();
      fileINI = new INIFileProperties();
      fileINI.load(fileName);

      tabNames = new String[fileINI.size()];
      tabs = new JPanel[fileINI.size()];
      subCounter = new Integer[fileINI.size()];
      for (Enumeration e = fileINI.keys(); e.hasMoreElements();) {
         Object key = e.nextElement();
         tabNames[counter] = key.toString().trim();
         tabs[counter] = new JPanel();
         try {
            prop = (Properties) fileINI.get(key);
            subCounter[counter]= new Integer(prop.size());
            subTotal += subCounter[counter].intValue();
         }
         catch (ClassCastException ex) {System.err.println(ex);}
         catch (NullPointerException ex) {System.err.println(ex);}
         counter++;
      }
      counter = 0;
      valueNames = new String[subTotal];
      values = new String[subTotal];
      valueFields = new JTextField[subTotal];
      for (Enumeration e = fileINI.keys(); e.hasMoreElements();) {
         Object key = e.nextElement();
         tabs[counter].setLayout(new GridBagLayout());
         GridBagConstraints c = new GridBagConstraints();
         c.insets = new Insets(3, 3, 3, 3);
         c.gridx = GridBagConstraints.RELATIVE;
         c.gridy = 0;
```

```
            try {
               prop = (Properties) fileINI.get(key);
               int i = 0;
               for (Enumeration en = prop.keys(); en.hasMoreElements();) {
                  Object key2 = en.nextElement();
                  valueNames[valueCounter] = key2.toString().trim();
                  values[valueCounter] = new
                    String(prop.get(key2).toString().trim());
                  valueFields[valueCounter] = new JTextField(22);
                  valueFields[valueCounter].setForeground(Color.black);
                  valueFields[valueCounter].setText(values[valueCounter]);
                  if (i != 0 && i%2 == 0) {
                     c.gridy++;
                  }
                  JLabel label = new JLabel(valueNames[valueCounter]);
                  label.setForeground(Color.blue);
                  tabs[counter].add(label, c);
                  tabs[counter].add(valueFields[valueCounter], c);
                  i++;
                  if (i> subCounter[counter].intValue()){
                     i =0;
                  }
                  valueCounter++;
               }
               pane.addTab(tabNames[counter], tabs[counter]);
               counter++;
            }
            catch (ClassCastException ex) {System.err.println(ex);}
            catch (NullPointerException ex) {System.err.println(ex);}
         }
         this.add(pane);

      }
      public String getFileName() {return fileName;}
      public String[] getTabNames() {return tabNames;}
      public String[] getValueNames() {return valueNames;}
      public String[] getValues() {return values;}
      public Integer[] getSubCounter() {return subCounter;}
      public JTextField[] getValueFields() { return valueFields; }

}


// The INIFileWriter class

package nsfssim;
/**
  * This class writes an INI file.
**/
import java.io.*;
import java.util.*;
import javax.swing.*;

public class INIFileWriter {
```

```
// instance variables
   private String fileName;
   private Properties hash;

// constructors
   public INIFileWriter(String file, String[] bracket, Integer[]
     keyNums, String[] keys, String[] values ) {
      hash = new Properties();
      Properties hash2;
      fileName = file;
      String nullString = null;
      int k = 0;
      for (int i =0; i < bracket.length; i++) {
         hash2 = new Properties();
         for (int j = 0; j < keyNums[i].intValue(); j++) {
            hash2.put(keys[k], values[k]);
            k++;
         }
         hash.put(bracket[i], hash2);
      }
      this.checkFile();
   }

   public INIFileWriter(String file, Properties prop) {
      hash = (Properties) prop.clone();
      fileName = file;
      this.checkFile();
   }

// instance methods
   public void checkFile() {
      File file = new File(fileName);
      if (file.exists()) {
         JFrame f = new JFrame("Overwrite?");
         int result = JOptionPane.showConfirmDialog(f,
           new String("Save " + file.toString() + "?"), "Save File",
           JOptionPane.YES_NO_OPTION);
         if (result == JOptionPane.YES_OPTION) {
            f.dispose();
            this.makeFile();
            this.showSavedMessage();
         }
         else {
            f.dispose();
         }
      }
   }

   public void makeFile() {
      StringBuffer buf = new StringBuffer();
      try {
         PrintWriter printOut = new PrintWriter(new
           FileWriter(fileName));
         Properties prop = null;
         for (Enumeration e = hash.keys(); e.hasMoreElements(); ) {
            Object key = e.nextElement();
```

```
            buf.append('\n');
            buf.append('[');
            buf.append(key.toString());
            buf.append(']');
            buf.append('\n');
            try {
               prop = (Properties) hash.get(key);
               for (Enumeration f = prop.keys(); f.hasMoreElements();)
               {
                  Object key2 = f.nextElement();
                  Object value2 = prop.get(key2);
                  buf.append(key2.toString());
                  buf.append(" = ");
                  if (value2 != null){
                     if (value2.toString().equals("null")) {}
                     else {
                        buf.append(value2.toString());
                     }
                  }
                  buf.append('\n');
               }
            }
            catch (ClassCastException ex)
              {buf.append(hash.get(key).toString());}
            catch (NullPointerException ex) {}
         }
           printOut.print(buf);
           printOut.flush();
           printOut.close();
      }
      catch (IOException e) { System.out.println(e);}
   }

   public void showSavedMessage() {
      JFrame f = new JFrame("Saved");
      JOptionPane.showMessageDialog(f, new String("File " + fileName +
        " saved."), "Save Complete", JOptionPane.INFORMATION_MESSAGE);
      f.dispose();
   }

   public String getName() {
      return fileName;
   }

}
```

// The JPanelDialog class

```
package nsfssim;

import simkit.util.*;

import java.util.*;
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class JPanelDialog extends JDialog implements
  PropertyChangeListener {

   protected JOptionPane optionPane;
   private static String[] options = { "Cancel" , "OK"};
   private static String okString = "OK";
   private JTextField[] fields;

   public JPanelDialog(Frame f, String title, boolean modal, JPanel
    panel, JTextField[]
     textFields, String words) {
      super(f, title, modal);
      fields = new JTextField[textFields.length];
      for (int i=0; i < textFields.length; i++) {
         fields[i] = textFields[i];
      }
      Object[] message = new Object[] { panel, words };
      optionPane = new JOptionPane(
        message, JOptionPane.PLAIN_MESSAGE,
        JOptionPane.OK_CANCEL_OPTION
      );
      optionPane.addPropertyChangeListener(this);
      this.getContentPane().add(optionPane, BorderLayout.CENTER);
      this.pack();
      this.setLocationRelativeTo(f);
      this.setResizable(false);
   }

// This method gets the result of the dialog
   public String getValue() {
      String selectedValue = null;
      StringBuffer returnValue = new StringBuffer();
      if (optionPane.getValue()!= JOptionPane.UNINITIALIZED_VALUE) {
         int result = ((Integer) optionPane.getValue()).intValue();
         if (result == JOptionPane.OK_OPTION) {
            for (int i = 0; i < fields.length; i++) {
               String text = fields[i].getText().trim().replace(' ',
                 '_');
               if (text.equals("")) {
                  returnValue.append("null");
               }
               else {
                  returnValue.append(text);
               }
               returnValue.append(" ");
            }
            selectedValue = returnValue.toString().trim();
         }
      }
      return selectedValue;
   }
```

```
// PropertyChangeListener for JOptionPane
   public void propertyChange(PropertyChangeEvent evt) {
      if (evt.getPropertyName().equals(JOptionPane.VALUE_PROPERTY)) {
         this.dispose();
      }
   }

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: CREATING ANIMATION IN NSFSSIM

NSFSSim is structured with a Model-View-Controller (MVC) design. That is, a non-visual simulation model (utilizing Simkit components and classes written in Java) operates independently of the visual view (e.g., NSFSSim's animation mode); a controller (in the form of keyboard and mouse events applied to the GUI) serves to synchronize the model with the view. The model and the view do not have to be aware of the existence of the other to function properly.

When one clicks on NSFSSim's main window to enable the animation mode, two things occur: an animation window opens and a "Ping" thread is enabled. When this thread is enabled, a "Ping" event is placed on the event list at regular intervals (the user can modify the interval between the "Ping" events). When each "Ping" event occurs, the Movers in the simulation model are painted in the animation window. Two of the classes written to create the animation in NSFSSim are `PingThread` and `AnimationFrame`, which are listed here in their entirety:

```
// The PingThread class

package nsfssim;

import simkit.*;

import java.awt.event.*;
import javax.swing.*;
import java.lang.reflect.*;

/**
 *  <P> An extremely simple way to animate Simkit programs.
 *  a Ping event occurs every deltaT utints of simulated time, which
 *  correspond roughly to millisPerSimTime milliseconds of real time
 *  (your mileage may vary).  Any listeners to Ping may do as they
 *  wish, such as updating the position of units drawn on a screen.
 *
 *  <P> This is perhaps an overly naive approach.  Suggestions are
```

```
 *  welcome.
 *
 *  @author Arnold Buss
**/

public class PingThread extends SimEntityBase implements Runnable {

   private double deltaT;           // Time between Pings events
   private double millisPerSimtime; // Real time per simulated time
   private boolean pinging;         // true if currently active

   private long realTimeStep;
   private long startStep;

// constructors
   public PingThread(double dt, double mpst) {
      this.setDeltaT(dt);
      this.setMillisPerSimtime(mpst);
   }

   public PingThread(double dt, double mpst, boolean pinging) {
      this(dt, mpst);
      this.setPinging(pinging);
   }

/**
 *  Simkit initialization -- if instance is created with
 *  <CODE>pinging</CODE> set true, then create Thread and start it.
**/
   public void doRun() {
      if (this.isPinging()) {
         this.startPinging();
      }
   }

/**
 *  Start pinging and wait forever (or until the Thread is stopped).
 *  The <CODE>while</CODE> loop appears necessary to keep the Thread
 *  from terminating by returning from <CODE>run()</CODE>.
**/
    public void startPinging() {
       new Thread(this).start();
    }

    public void run() {
       this.setPinging(true);
       waitDelay("Ping", 0.0);
       startStep = System.currentTimeMillis();
    }

/**
 *  Stop and shut down the Event List.
**/
   public void stopPinging() {
      this.setPinging(false);
      this.interruptAll();
```

```
    }

/**
 *  The main point of the class is the Ping event, which actually does
 *  nothing in and of itself other than schedule the next Ping event.
 *  Note that the sleep time is the number of milliseconds equivalent
 *  to deltaT, as specified by the user.
**/
    public synchronized void doPing() {
        if (isPinging()) {
            waitDelay("Ping", deltaT);
            try {
                Thread.sleep((long) (deltaT * millisPerSimtime));
            }
            catch (InterruptedException e) {}
        }
        long now = System.currentTimeMillis();
        realTimeStep = now - startStep;
        startStep = now;
        long offBy = realTimeStep - (long) (deltaT * millisPerSimtime);
    }

    public void pause() { Schedule.pauseSimulation(); }
    public void resume() { this.startPinging(); }

    public void setDeltaT(double dt) {deltaT = dt;}
    public void setMillisPerSimtime(double mpst) {millisPerSimtime =
      mpst;}
    public void setPinging(boolean p) {pinging = p;}

    public double getDeltaT() {return deltaT;}
    public double getMillisPerSimtime() {return millisPerSimtime;}
    public boolean isPinging() {return pinging;}

}
```

// The AnimationFrame class

```
package nsfssim;
/**
 *  <P> This class paints Movers when Ping events occur.
 *  @author H.B. Le
**/
import simkit.*;
import simkit.smd.*;

import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.text.*;

public class AnimationFrame extends JFrame implements
```

```
  SimEventListener {

   private static String DEFAULT_TITLE;
   private static Icon BACKGROUND;

   static {
      DEFAULT_TITLE = "NSFSSim Animation";
      BACKGROUND = new ImageIcon(AnimationFrame.class.
        getResource("icons/geo.gif").getFile());
   }

// instance variables
   private Icon area;                  // The background image
   private Hashtable2 icons;           // Stores the icon file names
   private Vector entities;            // This instance's movers
   private Image offscreen;            // For double-buffering the display
   private JPanel sandbox;
   private Graphics dbuf;
   private PingThread pt;              // This instance's PingThread

// constructors
   public AnimationFrame(PingThread ping) {
      this(DEFAULT_TITLE, ping);
   }

   public AnimationFrame(String title, PingThread ping) {
      super(title);
      pt = ping;
      this.init();
      area = BACKGROUND;
   }

   public AnimationFrame(int x, int y, int w, int h, PingThread ping) {
      this(x, y, w, h, ping, BACKGROUND);
      this.setBounds(x, y, w, h);
      pt = ping;
      this.init();
   }

   public AnimationFrame(int x, int y, int w, int h, PingThread ping,
     Icon geo) {
      super(DEFAULT_TITLE);
      this.setBounds(x, y, w, h);
      pt = ping;
      area = geo;
      this.init();
   }

   public AnimationFrame(String title, int x, int y, int w, int h,
    PingThread ping, Icon geo, Hashtable2 theIcons) {
      super(title);
      this.setBounds(x, y, w, h);
      pt = ping;
      area = geo;
      this.init();
      this.setIcons(theIcons);
```

```
    }

// instance methods
    public void init() {
        sandbox = new JPanel();
        sandbox.setBackground(Color.white);
        this.getContentPane().add(sandbox, BorderLayout.CENTER);
        pt.addSimEventListener(this);
        this.getContentPane().add(new PingPanel(pt, this),
          BorderLayout.SOUTH);
        entities = new Vector();
    }

/**
 *  Redraw the screen based on the current position of the Movers using
 *  double-buffering.
**/
    protected void updateEntities() {
        Graphics g = sandbox.getGraphics();
        if (offscreen == null) {
            offscreen = sandbox.createImage(sandbox.getSize().width,
              sandbox.getSize().height);
        }
        dbuf = offscreen.getGraphics();
        dbuf.fillRect(0, 0, getContentPane().getSize().width,
          getContentPane().getSize().height);
        area.paintIcon(getContentPane(), dbuf, 0, 0);
        for (Enumeration e = entities.elements(); e.hasMoreElements();) {
            Mover nextMover = (Mover) e.nextElement();
            if (nextMover instanceof NSFSShip) {
                this.paintShipGraphic(nextMover);
            }
            else if (nextMover instanceof ArtilleryBattery &&
              ((ArtilleryBattery) nextMover).isAlive()) {
                this.paintBatteryGraphic(nextMover);
            }
            else if (nextMover instanceof ArtilleryBattery &&
              !((ArtilleryBattery) nextMover).isAlive()) { }
            else {
                int x = (int) nextMover.getCurrentLocation().getXCoord();
                int y = (int) nextMover.getCurrentLocation().getYCoord();
                this.paintGraphic(this.getIcon(nextMover, "default"), x,
                  y);
            }
        }
        g.drawImage(offscreen, 0, 0, this);
        g.dispose();
        dbuf.dispose();
    }

    /**
     *  Paints the ship using one of several possible icons.
     *  @param ship = the ship to be painted
    **/
    public void paintShipGraphic(Mover ship) {
        int x = (int) ship.getCurrentLocation().getXCoord();
```

```
      int y = (int) ship.getCurrentLocation().getYCoord();
      if (((NSFSShip) ship).isMoving()) {
         if (((NSFSShip) ship).isCommencingOnloadSequence()) {
            this.paintGraphic(this.getIcon(ship, "stbdUW"), x, y);
         }
         else {
            this.paintGraphic(this.getIcon(ship, "portUW"), x, y);
         }
      }
      else if (((NSFSShip) ship).isCommencingOnloadSequence()) {
         this.paintGraphic(this.getIcon(ship, "stbd"), x, y);
      }
      else {
         this.paintGraphic(this.getIcon(ship, "port"), x, y);
      }
   }

   /**
    *  Paints the battery using one of several possible icons.
    *  @param battery = the battery to be painted
   **/

   public void paintBatteryGraphic(Mover battery) {
      int x = (int) battery.getCurrentLocation().getXCoord();
      int y = (int) battery.getCurrentLocation().getYCoord();
      if (!((ArtilleryBattery) battery).isFiring()) {
         if (((ArtilleryBattery) battery).isAtFullStrength()) {
            this.paintGraphic(this.getIcon(battery, "fullStrength"), x,
              y);
         }
         else if (((ArtilleryBattery) battery).getCurrentNumberGuns() >
           1) {
            this.paintGraphic(this.getIcon(battery, "weak"), x, y);
         }
         else {
            this.paintGraphic(this.getIcon(battery, "neardead"), x, y);
         }
      }
      else {
         this.paintGraphic(this.getIcon(battery, "firing"), x, y);
      }
   }

   public void paintGraphic(Icon icon, int x, int y) {
      icon.paintIcon(getContentPane(), dbuf, x, y);
   }


/**
 *  Adds a new mover.
 *  @param m = the new Mover added.
**/
   public void addMover(Mover m) {
      if (!entities.contains(m)){
         entities.addElement(m);
      }
```

```
    }

/**
 *  Removes a mover.
 *  @param m = the removed Mover.
**/
    public void removeMover(Mover m) {
        if (entities.contains(m)) {
            entities.removeElement(m);
        }
    }

/**
 *  Removes all movers.
**/
    public void removeMovers() {
        entities.clear();
    }

/**
 *  Gets a copy of movers in a thread-safe manner.
**/
    public Vector getMovers() {
        Vector copy = null;
        synchronized(entities) {
            copy = (Vector) entities.clone();
        }
        return copy;
    }

/**
 *  Here's where the Ping event is heard and entities are updated.
**/
    public void processSimEvent(SimEvent e) {
        if (e.getEventName().equals("Ping")) {
            this.updateEntities();
        }
    }

    public void setIcons(Hashtable2 theIcons) { icons = theIcons; }

    public Icon getIcon(Mover mover, String iconKey) {
        String moverClass = mover.getClass().getName();
        String iconFile = icons.get(moverClass, iconKey).toString();
        return new
          ImageIcon(AnimationFrame.class.getResource(iconFile).
            getFile());
    }

    public JPanel getSandbox() { return sandbox; }
    public PingThread getPingThread() { return pt; }

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Allen, D.J., et. al., *Naval Surface Fire Support Road Map Study, Phase 1 Report*, Johns Hopkins University Applied Physics Laboratory Report, VS-96-005, October 1996.

2. Bradley, Gordon H. and Arnold H. Buss, *An Architecture for Dynamic Planning Systems Using Loosely Coupled Components*, Proposal for Reimbursable Research, Operations Department, Naval Postgraduate School, 1997.

3. Chien, Stanfield L., *Optimizing Ordnance Loadout of Navy Surface Combatants Operating in Support of Naval Surface Fire Support*, Master's Thesis, Operations Research Department, Naval Postgraduate School, 1997.

4. Foss, Christopher F., *Jane's Armour and Artillery, Nineteenth Edition 1998-1999*, Alexandria: Jane's Information Group, Inc., 1998.

5. Geary, David M., *Graphic Java 2, Volume II: Swing*, Palo Alto: Sun Microsystems, Inc., 1999.

6. Holzer, Robert, "ERGM Complexity Prompts Independent Review Panel," *Defense News*, August 2, 1999.

7. Horstmann, Cay S. and Gary Cornell, *Core Java 1.1, Volume 1 - Fundamentals*, Mountain View: Sun Microsystems, Inc., 1997.

8. Law, Averill M. and David W. Kelton, *Simulation Modeling and Analysis*, New York: McGraw-Hill, Inc., 1991.

9. Office of the Chairman of the Joint Chiefs of Staff, *Department of Defense Dictionary of Military and Associated Terms*, Joint Pub 1-02, March 23, 1994, as amended through April 6, 1999.

10. Program Executive Officer, Cruise Missiles Project and Unmanned Aerial Vehicles Joint Project, *Tomahawk Weapon System Baseline IV Phase 1: Tactical Tomahawk Concept of Operations Document*, JCM-2237 (Draft), September 24, 1998.

11. Schweizer, Roman, "LASM-NTACMS Duel Heats Up in Pentagon, Issue Could Head to JROC," *Inside the Navy*, March 22, 1999.

12. Schweizer, Roman, "Navy Picks Land Attack Standard Over Army Missile to Outfit Aegis Ships", *Inside the Navy*, May 4, 1998.

13. Stork, Kirk, *Sensors in Object Oriented Discrete Event Simulation*, Master's Thesis, Operations Research Department, Naval Postgraduate School, 1996.

14. Townsend, James R., *Defense of Naval Task Forces From Anti-Ship Missile Attack*, Master's Thesis, Operations Research Department, Naval Postgraduate School, 1999.

15. Zimm, A.D., et. al., *Land Attack Warfare Technical Studies*, Johns Hopkins University Applied Physics Laboratory Report, JWR-98-013, February 1998.

16. Zimm, A.D., *Advanced Gun Study: Effectiveness Analyses: TAFSM, ELAN, and ARTQUIK Modeling*, Johns Hopkins University Applied Physics Laboratory Report, JWR-99-001, Revision 1, February 17, 1999.

17. Zimm, A.D., *Advanced Gun Study: Supplemental Analysis: Land Attack Effectiveness*, Johns Hopkins University Applied Physics Laboratory Report, JWR-99-007, May 16, 1999.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Fort Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Mr. Stephen M. Biemer 2
   Joint Warfare Analysis Department, JHU/APL
   Johns Hopkins Rd.
   Laurel, Maryland 20723-6099

4. Mr. John F. Keane 1
   Joint Warfare Analysis Department, JHU/APL
   Johns Hopkins Rd.
   Laurel, Maryland 20723-6099

5. Mr. Richard L. Miller 1
   Joint Warfare Analysis Department, JHU/APL
   Johns Hopkins Rd.
   Laurel, Maryland 20723-6099

6. Mr. Edward A. Smyth 1
   Joint Warfare Analysis Department, JHU/APL
   Johns Hopkins Rd.
   Laurel, Maryland 20723-6099

7. Mr. Alan D. Zimm 1
   Joint Warfare Analysis Department, JHU/APL
   Johns Hopkins Rd.
   Laurel, Maryland 20723-6099

8. Professor Arnold H. Buss, Code OR/Bu 2
   Department of Operations Research
   Naval Postgraduate School
   Monterey, California 93943-5000

9. LCDR Douglas J. MacKinnon, Code OR/Mg 1
   Department of Operations Research
   Naval Postgraduate School
   Monterey, California 93943-5000

10. LT Hung B. Le 2
2565 Archdale Dr.
Virginia Beach, Virginia 23456-6881